Ollscoil na hÉireann, Corcaigh

National University of Ireland, Cork

A Holistic Architecture using Peer to Peer (P2P) Protocols for the Internet of Things and Wireless Sensor Networks

Thesis presented by

David Tracey, B.E., M.Sc

for the degree of

Doctor of Philosophy

University College Cork

School of Computer Science and Information Technology

Head of School: Prof. Cormac Sreenan

Supervisor(s): Prof. Cormac Sreenan

February 2020

Contents

List of Figures vii
List of Tables ix
Declaration xi
Acknowledgements xii
Abstract xiii
Nomenclature xv
Introduction 1
1.1 Thesis Statement 1
1.2 Background 1
1.3 Motivation 4
1.4 Research Approach 5
1.5 Research Contributions 7
1.6 Publications 8
1.7 Thesis Structure 9
2 IoT and Wireless Sensor Networks 11
2.1 Introduction 11
2.2 Wireless Sensor Networks 12
2.3 Wireless Sensor Devices 13
2.4 Wireless Sensor Network Stack 16
2.4.1 Wireless Interface Layers 17
2.4.2 Network and Routing Layers 19
2.4.3 Wireless Sensor Network Routing 20
2.4.4 Wireless Sensor Network Application Layer Protocols 24
2.5 Wireless Sensor Network Information Models 26
2.5.1 IPSO and OMA LWM2M 26
2.5.2 Common Information Model (CIM) 28
2.5.3 SensorML 29
2.6 Web of Things (WoT) 30
2.7 Characteristics of IoT Applications using Wireless Access 34
2.8 Summary 37
3 Distributed System Concepts 39
3.1 Introduction 39

3.2 Architectural Approaches 39
3.2.1 RESTFul Architecture Style 39
3.2.2 Middleware Approaches 40
3.2.3 Autonomic and Cognitive Architectures 43
3.3 Big Data and NoSQL Approaches 44
3.4 Cloud and Sensor Platforms 45
3.5 Fog and Edge Computing 46
3.6 Tuple Approaches 51
3.7 Peer To Peer (P2P) Systems 52
3.7.1 Freenet 55
3.7.2 JXTA 56
3.7.3 Distributed Hash Table Based Networks 57
3.7.4 BitTorrent 66
3.8 P2P in Wireless Sensor Networks 69
3.8.1 WSN as a Peer via a Gateway to Mobile Network 69
3.8.2 Distributed Hash Tables in WSNs 70
3.8.3 Tiered Chord (TChord) 71
3.8.4 Service and Resource Discovery using P2P 72
3.8.5 TinyTorrents 73
3.9 Cache Algorithms 74
3.9.1 Use Based Cache Algorithms (LRU, LFU, MRU) 76
3.9.2 Web Cache Approaches 77
3.9.3 Paging Algorithms 77
3.9.4 Summary of Cache Algorithm Performance 80
3.9.5 Caching in Wireless Sensor Networks 80
3.10 Summary 81
4 Analysis, Architecture and Design 83
4.1 Analysis 84
4.1.1 Architectural Lessons 84
4.1.2 Architectural Requirements 85
4.1.3 WSN Software Development Requirements 87
4.1.4 System Model 88
4.1.5 Security Considerations 89
4.1.6 Findings from the Review of Prior Work 90
4.2 CacheL Algorithm 96
4.3 Holistic Architecture and Abstractions 102

4.3.1 Service Abstractions 105
4.3.2 Object Space 108
4.3.3 Local Instrumentation (li) Layer 109
4.4 HPP Overlay using Distributed Hash Table 110
4.5 HPP Protocol Design 112
4.5.1 HPP Forwarding and Routing 113
4.5.2 HPP Actors 114
4.5.3 HPP Message Header 117
4.5.4 HPP Message Types (or Commands) 119
4.5.5 HPP Message Flows 122
4.5.6 Decentralised Control 125
4.5.7 Energy Consumption 126
4.5.8 Resource Discovery 127
4.6 Summary 128
5 Implementation 129
5.1 Linux and Contiki Implementations 131
5.2 HPP Implementation 132
5.2.1 Channel and Endpoint Communication Layers 132
5.2.2 HPP Message Processing 134
5.2.3 DHT Implementation 136
5.3 Data Model Implementations 137
5.3.1 Data Model Service Layer 139
5.3.2 Local Instrumentation Layer 140
5.3.3 Integration with Erbium CoAP Implementation 142
5.3.4 Contiki Implementation of OMA LWM2M 144
5.4 HBase Integration 147
5.5 CacheL Implementation 149
5.6 Summary 151
6 Experimental Evaluation 153
6.1 Consideration of Architectural Requirements 154
6.2 Architectural Comparison 155
6.3 Example Scenarios 157
6.4 Linux and Contiki Implementation 161
6.5 Use of Abstractions 163
6.5.1 Data Models 164
6.5.2 HBase Integration 165

6.6 Mapping of OMA LWM2M and CIM Data Models 165
6.7 Memory Use 167
6.8 HPP Performance 170
6.8.1 Node Functionality for Testing 170
6.8.2 Constrained Device (Contiki) Tests 173
6.8.3 Linux Node Tests 177
6.9 CacheL 192
6.9.1 Implementation Complexity 193
6.9.2 Performance Comparison of LRU and CacheL using YCSB 193
6.9.3 Optimisations to CacheL 197
6.9.4 Performance Characteristics of CacheL 197
6.10 Summary 201
7 Conclusion 203

List of Figures

Figure 1 The IoT leading to “Systems of Systems” [3] 3
Figure 2 Tmote Sky Device [22] 15
Figure 3 Zolerta Z1 Device [23] 15
Figure 4 Network Stack including RPL 17
Figure 5 OMA LWM2M and IPSO 27
Figure 6 Abstract Architecture of W3C WoT [59] 32
Figure 7 Eclipse IoT Stacks [69] 41
Figure 8 OpenFog Architecture Scenario [96] 50
Figure 9 OpenFog Reference Architecture Description with Perspectives [96] ... 50
Figure 10 Example of Lookup in Chord 60
Figure 11 Kademlia Binary Tree Example 63
Figure 12 Clock Algorithm 78
Figure 13 CacheL Algorithm Lists Sweep 99
Figure 14 Node Architecture 104
Figure 15 Holistic Architecture 107
Figure 16 Sample HPP Service Interaction 124
Figure 17 Resource Access over HPP and CoAP 138
Figure 18 Local Instrumentation Structures 141
Figure 19 YCSB Test Setup 150
Figure 20 MQTT Bridging 158
Figure 21 Federation of MQTT using HPP 158
Figure 22 Vehicle Data Hub Architecture 159
Figure 23 OpenFog Transportation:Smart Car & Traffic Control System [96] .. 160
Figure 24 Smart Transport Fog System with HPP 161
Figure 25 Sample of Test Messages for Direct Test Node 173
Figure 26 Simulation with 5 nodes 175
Figure 27 Simulations with 7 nodes and 9 nodes 175
Figure 28 Single Bootstrap Node Test Scenario 178
Figure 29 Multiple Bootstrap Peer Test Scenario 179
Figure 30 Request Processing Times on Bootstrap Peers 183
Figure 31 Request Processing times for Multiple (50 or 100) Bootstrap Peers .. 183
Figure 32 Times to Receive Replies on Non-bootstrap Peers 184
Figure 33 Counts of key variables for Cache Size 100, Lease 0-100ms 198
Figure 34 Backhand Delete Counts per Cache Size, Lease 0-100ms 199
Figure 35 Lease Expired Counts per Cache Size, Lease 0-100ms 199
Figure 36 Lease Counts for Cache Size 100, Lease 0-100ms 200

List of Tables

Table 1 Memory Usage of CIM Implementation 167
Table 2 Memory Usage of IPSO Implementation 168
Table 3 Memory Use of Nodes of Different Capability 169
Table 4 HPP Message Processing Times for 5 and 9 node tests (except node 5) 176
Table 5 Variability of Get Message Times (ms) in 20 Node tests 185
Table 6 Number of Requests handled by Bootstrap Peers 186
Table 7 Counts for Bootstrap Peers 189
Table 8 Counts for Non-bootstrap Peers 190
Table 9 Counts for "1bs-direct-test" with Termination (kill) 192
Table 10 20 Peer "1bs-direct-test" Hello times with and without Termination .. 192
Table 11 YCSB Workloads 193
Table 12 LRU and CacheL (Without Leases) Hit Ratios per Workload 195
Table 13 CacheL (With Leases) Hit Ratios per Workload 196

Declaration

This is to certify that the work I am submitting is my own and has not been submitted for another degree, either at University College Cork or elsewhere. All external references and sources are clearly acknowledged and identified within the contents. I have read and understood the regulations of University College Cork concerning plagiarism.

David Tracey

February 2020

Acknowledgements

I would like to thank my supervisor Professor Cormac Sreenan for his guidance on research and reviewing of draft papers, as well as for brainstorming ideas with me during the course of this work. In addition to this, his support, understanding and encouragement were very important given this work was performed on a part-time basis. I also appreciate his enthusiasm for me to explore a wide range of research as part of this work.

Finally, I am so grateful to my wife Fiona and my daughters, Aisling, Hannah and Sinead for their support, including making all those cups of coffee that were needed along the way.

Abstract

Wireless Sensor Networks (WSNs) interact with the physical world using sensing and/or actuation. The wireless capability of WSN nodes allows them to be deployed close to the sensed phenomenon. Cheaper processing power and the use of micro IP stacks allow nodes to form an "Internet of Things" (IoT) integrating the physical world with the Internet in a distributed system of devices and applications. Applications using the sensor data may be located across the Internet from the sensor network, allowing Cloud services and Big Data approaches to store and analyse this data in a scalable manner, supported by new approaches in the area of fog and edge computing. Furthermore, the use of protocols such as the Constrained Application Protocol (CoAP) and data models such as IPSO Smart Objects have supported the adoption of IoT in a range of scenarios.

IoT has the potential to become a realisation of Mark Weiser's vision of ubiquitous computing where tiny networked computers become woven into everyday life. This presents the challenge of being able to scale the technology down to resource-constrained devices and to scale it up to billions of devices. This will require seamless interoperability and abstractions that can support applications on Cloud services and also on node devices with constrained computing and memory capabilities, limited development environments and requirements on energy consumption.

This thesis proposes a holistic architecture using concepts from tuple-spaces and overlay Peer-to-Peer (P2P) networks. This architecture is termed as holistic, because it considers the flow of the data from sensors through to services. The key contributions of this work are: development of a set of architectural abstractions to provide application layer interoperability, a novel cache algorithm supporting leases, a tuple-space based data store for local and remote data and a Peer to Peer (P2P) protocol with an innovative use of a DHT in building an overlay network. All these elements are designed for implementation on a resource constrained node and to be extensible to server environments, which is shown in a prototype implementation. This provides the basis for a new P2P holistic approach that will allow Wireless Sensor Networks and IoT to operate in a self-organising ad hoc manner in order to deliver the promise of IoT.

Nomenclature

6LoWPAN IPv6 over Low power Wireless Personal Area Networks.

API Application Programming Interface.

BAN Body Area Network.

BLE Bluetooth Low Energy.

CoAP Constrained Application Protocol.

CoRE Constrained RESTFul Environments.

HATEOAS Hypermedia as the Engine of Application State.

HTTP Hypertext Transfer Protocol.

IANA Internet Assigned Numbers Authority

IETF Internet Engineering Task Force.

IoT Internet of Things.

IPSO IP for Smart Objects.

JVM Java Virtual Machine.

LLN low-power lossy network.

LWM2M OMA Lightweight M2M.

M2M machine-to-machine.

MQTT Message Queuing Telemetry Transport.

OMA Open Mobile Alliance.

PaaS Platform as a Service.

PAN Personal Area Network

REST Representational State Transfer.

RFID Radio-Frequency Identification.

RPL IPv6 Routing Protocol for Low-Power and Lossy Networks.

SaaS Software as a Service.

TLS Transport Layer Security.

URI Uniform Resource Identifier.

WoT Web of Things.

WSN wireless sensor network.

Introduction

"Architecture starts when you carefully put two bricks together.
There it begins", Ludwig Mies van der Rohe, June 1959

This dissertation is concerned with creating a holistic architecture that provides simple, powerful abstractions for the end-to-end flow of data from constrained Wireless Sensor Network (WSN) devices to consuming services, possibly Cloud based in an Internet of Things. This in line with the Cambridge English Dictionary definition of holistic as "dealing with or treating the whole of something or someone and not just a part". This architecture provides layers and abstractions that allow the provision of simple and consistent Application Programming Interfaces (APIs) to store and exchange sensor data. This architectural approach allows programmers to have a consistent approach in a variety of environments from software to send sensor data on constrained WSN devices to software for Cloud based services receiving, analysing and taking actions based on that data.

This chapter provides the background, motivation and approach taken for this work and states the contributions made.

1.1 Thesis Statement

This thesis demonstrates that a holistic architectural approach with a consistent set of abstractions, implementable on both constrained wireless sensor devices and as part of advanced services, can provide the flexibility, simplicity, interoperability and scalability required to realise the potential of the Internet of Things (IoT).

1.2 Background

There are many definitions of IoT and the possible scope of IoT can be seen in the definition from the ITU-T in Recommendation ITU-T Y.2060 of "a global

infrastructure for the information society, enabling advanced services by interconnecting (physical and virtual) things based on existing and evolving interoperable information and communication technologies" [1]. This Internet of Things (IoT) has been enabled by cheaper processing power, low power wireless and the use of micro IP stacks. This allows nodes to form an Internet of Things integrating the physical world with the Internet in a distributed system of devices and applications. Applications using the sensor data may be located across the Internet from the sensor network, allowing Cloud services and Big Data approaches to store and analyse this data in a scalable manner. This is supported by new approaches in the area of fog and edge computing to reduce the impact of network factors such as latency, intermittent connectivity and limited bandwidth by bringing at least some of the processing closer to the source of the data.

This IoT distributed system may include Wireless Sensor Networks (WSNs), where the wireless capability of WSN nodes allows them to be deployed close to the sensed phenomenon. There are a wide range of scenarios in which Wireless Sensor Networks are used or for which they are targeted [2]. The limits of wireless technologies in these scenarios compared to wired technologies are acceptable given the flexibility offered by the wireless connection, although the deployment of a wireless network may not be straightforward due to factors in the environment, such as physical obstacles.

Figure 1 from [3] shows the potential for IoT to move an existing product to a smart connected product, then a connected system to ultimately a "System of Systems" that links systems together. In the example given, a tractor company becomes part of a broader farm automation set of systems connecting farm machinery, irrigation systems, soil and nutrient sources with information on weather and crop prices.

Although the term IoT has been in existence for a number of years, the potential of new applications and services to take advantage of IoT is, however, limited by the difficulties imposed by the heterogeneous nature, constrained capabilities and the limited development environments of WSN nodes and the use of proprietary or specialized software/protocols. This is exacerbated by where the nodes are deployed, e.g. WSNs are often deployed in situations, such as remote locations, where they must be able to operate autonomously, or at least unattended, for long

periods of time. The subsequent trade-offs of device lifetime versus device processing capability versus ease of software development and ease of deployment, however, still remain for WSN devices and are often solved in a one-off and/or proprietary manner.

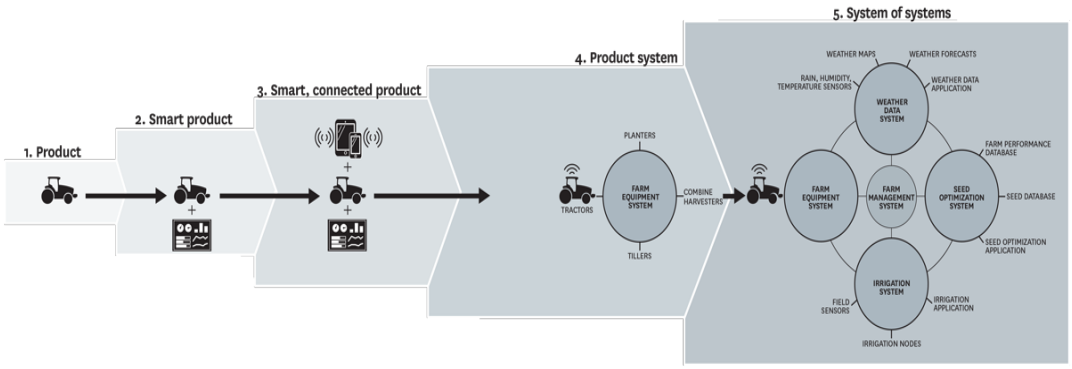

Figure 1 The IoT leading to “Systems of Systems” [3]

Without a few “killer applications” to drive volume and establish a number of large scale environments, IoT consists of a plethora of possible, separate solutions with distinct wireless technologies, operating systems for constrained devices, data models, architectural approaches and protocols or custom vendor solutions. Many solutions have limited, if any, interoperability beyond the use of the IP stack, slowing the growth and benefits that IoT could offer.

A report [4] highlights the importance of interoperability: “Of the total potential economic value the IoT enables, interoperability is required for 40 percent on average and for nearly 60 percent in some settings”. The potential to provide greater application interoperability can be seen in application layer protocols such as the Constrained Application Protocol (CoAP) [5], supplemented by the availability of data models such as IPSO Smart Objects [6] and the Open Mobile Alliance Lightweight Specification (OMA LWM2M) [7]. Similarly, the emerging area of fog computing should allow a service to be executed using components running in networks or different providers and this requires that fog and edge components be interoperable at the level of providers and architecture models and interfaces.

Despite such potential, however, it can be said in general that the Internet of Things currently uses numerous and distinct protocols and standards at all levels

of the communication stack and consists of a variety of purpose-built applications operating in a silo manner on network and system architectures often designed primarily for that application. Furthermore, these architectures may not be suited to wider use.

1.3 Motivation

The current situation in IoT with numerous and distinct protocols and standards at all levels of the communication stack is reminiscent of the early days of the adoption of Internet services, such as email. In that instance, the protocols and infrastructure for email were well understood with several implementations, but email use was generally limited to larger companies or academic institutions. It can be argued that it was the introduction of HTTP and particularly the Internet Browser, with supporting web-based email services, that drove the wider adoption of email. IoT is at a similar stage where it needs to break out of the limitations of its current situation.

The availability of increased storage and processing power at a lower cost with greater bandwidth has resulted in the rise of Cloud computing services and Big Data techniques, such as HBase and MapReduce in the Hadoop stack. This has allowed the creation of some IoT applications using the Cloud and the idea of "Sensing As A Service" [8]. Similar to the early days of Internet services, however, WSNs and IoT have not been able to take full advantage of Cloud and Big Data services. Hence, there are isolated end-to-end solutions for relatively small islands of sensor networks, instead of sensor data and services being universally and conveniently available, interoperable and scalable in the Cloud for consumers or even for developers.

A key challenge is whether the range of separate solutions with different approaches outlined earlier can enable the predicted growth in IoT to tens of billions of connected devices [9], both in terms of scaling the systems to store and analyse data, manage nodes and also in terms of developing services and node software in such a diverse environment. The scalability challenge requires not only being able to scale it up to billions of devices, but also to scale it down to resource-constrained devices in relatively small WSNs [10]. This requires seamless interoperability and a consistent set of abstractions and APIs/protocols,

particularly at the application layer to realise Mark Weiser's ubiquitous computing vision of tiny networked computers woven into everyday life [11].

This thesis is motivated by seeking to meet that key scalability and interoperability challenge and to consider which architectural approaches may accommodate the end-to-end flow of data and control in the Internet of Things in order to seamlessly include vast numbers of devices of varying capability and different WSN technologies into that distributed system.

1.4 Research Approach

There are a variety of possible IoT approaches and solutions including several micro-IP stacks on devices, a range of wireless protocols, a range of information models, different operating systems, different application protocols and custom/proprietary vendor solutions, e.g. for Cloud integration. These approaches generally target a particular scenario or a part of the end-to-end environment. Coupled with the lack of a dominant solution for hardware suppliers, device and application developers, this has hindered large-scale development, adoption and demand from potential customers. In contrast to this, the area of smartphone applications has flourished, even including the use of on-board sensors (or even external sensors) for applications such as fitness monitoring. In this case, the availability of a limited set of stable, well-known development environments and a large potential market has generated a wide range of applications and devices.

In order to create such an environment for IoT and to meet the key scalability and interoperability challenges outlined earlier, the approach in this work is to develop a holistic architecture, with a supporting set of consistent abstractions. This architecture is designed to meet a set of requirements for WSNs and IoT for the complete end-to-end flow of data. That flow of data includes when data is sent (and aggregated/stored/acted on) from constrained devices to border routers or edge gateways to Cloud services for storage or analysis. It also includes the flow in the opposite direction. This thesis considers the Internet of Things as consisting of a multiplicity of distributed systems and it reviews which architectural approaches may be suitable to meet the heterogeneity and scalability challenges outlined above. In particular, it considers the RESTFul architectural style and

BitTorrent as examples of systems that have achieved scale, for lessons to take from them in order to meet those requirements.

The research in this thesis also considers the following computing concepts in the context of an architecture to realise the potential of IoT:

- *Tuple Spaces* provide a repository of tuples (a finite list of elements) available for concurrent access into which producers post their tuples and from which consumers read those it wants. One benefit of using tuple spaces in a distributed system is the decoupling in time and space of tuple space communication, which enables interactions where applications can be added/removed independently.
- *Caching* provides the ability to aggregate and analyse data closer to the source (rather than relying solely on analysis in a data centre or Cloud service). It is becoming important in scenarios such as the development of fog and edge computing. The use of caches in a WSN seems an obvious approach to reduce the response time, use of bandwidth and to extend node battery life by reducing the number of hops to retrieve data. The overhead involved in caching and the limited node memory to hold cached data must, however, be considered.
- *Peer to Peer (P2P)* systems are mainly used for file sharing where their use is driven by advances in hard-disk capacity and bandwidth availability, P2P also has the potential to provide benefits in a general distributed system, such as decentralised control, support for nodes joining/leaving a group, robustness and scalability.
- *Distributed Hash Tables (DHT)* are used in a number of peer to peer systems to provide efficient routing, without centralized control, to nodes that send, receive, retrieve and forward information over a network, e.g. BitTorrent [12] and its use of Kademlia [13]. Indeed, DHTs “appear to provide a general-purpose interface for location-independent naming upon which a variety of applications can be built. Furthermore, distributed applications that make use of such an infrastructure inherit robustness, ease of operation, and scaling properties” [14].

A key part of the methodology adopted was to ensure that all the elements included in the holistic architecture could be implemented on constrained devices in a WSN, as well as on more powerful computing systems.

1.5 Research Contributions

A novel feature of this work is provided by its holistic architectural approach, where the same abstractions with a simple Peer to Peer (P2P) protocol are demonstrated running on constrained devices and more powerful servers. This holistic approach provides for the varied roles these entities may play in the flow of data from sensor to application, ranging from a simple source of a sensor reading to a Big Data store for that data. As such, the key contributions of this work are:

- development of a set of architectural abstractions to provide application layer interoperability, based on requirements derived from analysing WSNs and applications.
- a simple tuple space based datastore and API for both local and remote data that is able to hold data from information models such as the well-known OMA LWM2M and the Common Information Model (CIM).
- a novel, simple cache algorithm supporting leases, designed for constrained devices that provides a good cache hit ratio compared to the Least Recently Used (LRU) algorithm.
- a Peer to Peer (P2P) application layer protocol termed HPP (Holistic Peer to Peer) supporting this architecture.
- a novel use of a DHT in building an overlay network between peer nodes and in supporting data objects from remote devices.

All of these elements are designed for implementation on a resource constrained node and to be extensible to server environments. This work also presents a prototype implementation of the core architecture, cache algorithm, data store and P2P protocol. The intention of this work is to provide the basis for a new P2P holistic approach designed to allow Wireless Sensor Networks to operate in a self-organising ad hoc manner to deliver the promise of IoT, in contrast to the more common client-server and gateway approaches.

1.6 Publications

- D. Tracey, C. Sreenan, "A Holistic Architecture for the Internet of Things, Sensing Services and Big Data", Proceedings of the 13th IEEE/ACM International Symposium on Cluster, Cloud and Grid Computing, May 2013
  This paper presents an initial set of architectural requirements, a resulting layered architecture and abstractions for the data exchange roles taken by services on WSN nodes and in the Cloud. This forms the basis of the architectures considered in Chapter 3 and the architectural approach proposed in Chapters 4 and 5. It also outlines the initial proposal for the Holistic Peer to Peer (HPP) protocol described in its fully developed form in Chapters 5 and 6. It also presents an overview of an initial implementation, which is detailed in Chapter 6.
- D. Tracey, C. Sreenan, "CacheL - A Cache Algorithm using Leases for Node Data in the Internet of Things", Proceedings of the IEEE 4th International Conference on Future Internet of Things and Cloud (FiCloud-2016), August 2016 (*Awarded best paper at the conference*)
  This paper presents the contribution of the novel CacheL Algorithm suitable for constrained WSN nodes. This algorithm uses a cache replacement process inspired by the Clock algorithms for paging with the addition of lease management as part of the replacement process. Making lease management part of the cache replacement process removes the need for a separate periodic process to manage leases. This follows from the analysis of the role of caching in Chapter 4. The algorithm from the paper is described in detail in Chapter 5 and is evaluated in Chapter 6.
- D. Tracey, C. Sreenan, "OMA LWM2M in a Holistic Architecture for the Internet of Things", Proceedings of the IEEE 14th International Conference on Networking, Sensing and Control (ICNSC-2017), May 2017
  This paper demonstrates that the holistic architecture and its service-based abstractions, presented in Chapter 4, can support rich data models such as OMA LWM2M. It presents an integration of the tuple based store in Chapter 5 with an existing implementation of the OMA LWM2M model on the Contiki3.0 operating system. This demonstrated that the data-

centric holistic approach's set of abstractions were straightforward to implement even on a constrained node. This implementation and its subsequent development and evaluation are described in Chapters 5 and 6.

- D. Tracey, C. Sreenan, "How to see through the Fog? Using Peer to Peer (P2P) for the Internet of Things", Proceedings of the Globe-IoT 2019: 'Towards Global Interoperability among IoT Systems' jointly held with the IEEE 5th World Forum on Internet of Things, April 2019
  This paper proposes that meeting the challenges presented by IoT requires an architecture and a set of consistent abstractions for all components in the entire flow of data in IoT. In particular, it proposes that the fog computing architectures described in Chapter 3 must consider constrained devices as part of that flow. It also proposes that a P2P overlay network can be used to achieve scalability and high availability, especially at the edges of the Internet. It also considers which architectural approaches may be suitable for IoT and how they relate to fog computing. This is described in Chapters 3 and 4, which emphasise the lessons to be learned from the RESTFul architectural style and BitTorrent.
- D. Tracey, C. Sreenan, "Using a DHT in a Peer to Peer Architecture for the Internet of Things", Proceedings of the IEEE 5th World Forum on Internet of Things (WF-IoT), April 2019
  This paper presents the detail of a Holistic Peer-to-Peer (HPP) application layer protocol and its support for the data-centric approach in the holistic architecture, presented in Chapter 4. In particular, it considers the contribution of an application overlay using a Distributed Hash Table DHT, based on Kademlia described in Chapter 3. This overlay can span the WSN and services over the Internet, as well as being suitable for fog computing. The implementation using a DHT is detailed in Chapter 5 with an evaluation of results in Chapter 6.

1.7 Thesis Structure

This chapter has introduced the problem area being addressed and provided background information on Wireless Sensor Networks and IoT, including the potential impact, application areas and the challenges to be addressed. It has also introduced the motivation, approach taken and the technologies investigated in

this work. The contributions of this work have also been described. The rest of the dissertation is structured as follows:

Chapter 2 provides further background information and related work on IoT, Wireless Sensor Networks and Wireless Sensor Devices, including their information models

Chapter 3 provides background information and related work on distributed computing concepts that may be appropriate for use in IoT, including Big Data, tuple spaces, cloud and fog computing, peer-to-peer (P2P) systems and cache algorithms. It outlines related work in WSNs for these distributed computing concepts, particularly tuple spaces, P2P and cache algorithms.

Chapter 4 describes the design of the holistic architecture and its components, particularly the data model service layer, the local instrumentation layer, the object space layer, the CacheL algorithm and the Holistic Peer to Peer (HPP) Protocol itself.

Chapter 5 describes the methodology of the implementation of the holistic architecture components on Linux and Contiki, particularly the data model service layer, including its integration with the existing CoAP code on Contiki, the local instrumentation layer and the object space layer and library. It also presents the implementation of the CacheL algorithm and a HBase integration as an example of its integration with the Big Data solutions.

Chapter 6 presents the results of experiments showing the effectiveness of the abstractions in modelling CIM and OMA LWM2M objects, the relative simplicity and size of the code and the scalability and robustness of HPP. It also presents results related to the use of a Kademlia based DHT with HPP. This chapter also presents results from tests on the performance of the CacheL algorithm compared to LRU.

Chapter 7 concludes the dissertation with a summary of the work and describes the potential for future related work.

2 IoT and Wireless Sensor Networks

2.1 Introduction

There are myriad definitions of IoT, including definitions from standardisation or industry bodies such as ETSI, OneM2M, ITU, IETF, NIST, OASIS and W3C [15]. Perhaps the broadest definition that indicates the scope envisaged for IoT comes from the ITU-T in Recommendation ITU-T Y.2060 “a global infrastructure for the information society, enabling advanced services by interconnecting (physical and virtual) things based on existing and evolving interoperable information and communication technologies” [1]. Such definitions of IoT generally share the idea that IoT relates to the integration of the physical world with the virtual world of the Internet [16]. As such, IoT is characterised by an interconnected set of individually addressed and constrained devices in a distributed system, with sensing for physical phenomena and/or actuation capabilities. The growth in IoT is expected to result in tens of billions of connected devices [9] and consuming/controlling services that must be programmed and managed.

IoT’s use of the Internet allows the services using sensor data to be located across the Internet from the sensing device or WSN. This allows the use of Cloud services and Big Data approaches to provide the required scalability to store and analyse this data, supported by IoT services offered by Cloud providers. The volume of sensor data and network effects such as latency and bandwidth limits have, however, also resulted in the emergence of fog and edge computing to support some processing of the data closer to the device in order to reduce these effects and still retain the flexibility and scale of Cloud services for historical analysis or analysis across multiple WSNs.

This chapter contains an overview of the constituent elements of IoT including wireless sensor devices, wireless protocols and WSNs, the characteristics of

applications and information models in WSNs. It also presents overviews of the Web of Things and wireless technologies to illustrate the role of constrained devices and WSNs in the wider scope of IoT. Later chapters will provide an overview of distributed computing concepts such as Big Data, tuple spaces, Cloud and fog computing for sensor data and services. Those later chapters will also consider these in the context of IoT and particularly the WSN aspects, such as the constraints and opportunities that arise. For example. the wireless capability allows the sensor nodes to be deployed close to the phenomenon being observed, but their limited processing capability may limit their ability to interoperate with other nodes and their role in fog computing.

2.2 Wireless Sensor Networks

The increasing availability of sensors together with advances in wireless technologies and particularly cheaper processing power has allowed the emergence of Wireless Sensor Networks (WSNs). WSNs are based on using low-power radio chipsets to cover areas of interest with relatively inexpensive nodes. Combining WSNs with the use of IP has allowed nodes to form an “Internet of Things” (IoT), which is effectively a distributed system of devices and applications comprising sensing, computation and actuation. The role of sensing and WSNs in IoT can be seen in the following IEEE definition of IoT as “A network of items - each embedded with sensors which are connected to the Internet” [15]. WSNs have been deployed in a variety of application scenarios, such as environmental monitoring, surveillance and healthcare.

WSNs consist of nodes that sense a particular entity, collect (and possibly parse or aggregate) the data and send the data over a wireless link to one or more destinations and ultimately to an application, with particular requirements on QoS such as latency, jitter, bandwidth. This involves a number of interactions including:

- the sensing of the data
- the delivery of data to the application
- the sensor hardware interface to the radio or wireless layer on the node
- the Media Access Control (MAC) layer to provide access to the shared wireless resource

- the Network layer routing of the data across the WSN to the application, possibly using a gateway device from the WSN to an IP endpoint where the application resides.

The wireless capability allows the sensor nodes to be deployed close to the phenomenon being observed and their limited processing capability results in relatively low-cost devices so allowing a larger number of such sensors to be used. WSN deployments are, however, often dedicated and proprietary or specialised to optimise one particular aspect such as lifetime. WSNs are characterised by having a (possibly large) number of devices with sensing capabilities, limited processing capability and wireless connectivity to other nodes, such as another sensor node or a higher function gateway node.

It is important to point out, however, that a number of possible drawbacks to the use of wireless networks must be considered in a given scenario. For example, throughput. packet error rate, jitter and latency may experience significant fluctuations due to radio coverage, traffic load and interference. Also, there is a straightforward reduction in performance compared to a wired link, e.g. a short-range link carried by IEEE 802.15.4 technology will support only 250 kb/s. With this particular technology, only low to moderate amounts of data can reasonably be sent to a handheld or other portable device.

2.3 Wireless Sensor Devices

A wireless sensor device exists to sense one or more physical phenomenon, e.g. light, humidity, strain, voltage or temperature. It may also collect (and possibly parse or aggregate) the sensor data and send the data to one or more destinations and ultimately to an application. The device is also aware of the capabilities of the sensor in terms of the frequency at which it can sense, any delays or latencies in the sensing and the hysteresis required when interpreting readings. It is also aware whether thresholds (upper or lower) can be set as boundaries for it to report on. Rather than being a purely sensing device, it may also be an actuator, which determines whether an application purely requests data from the device or also has some control over device actuation. The device will also be aware of how often it can send data (not necessarily at the same as the frequency at which the sink application wants it).

The challenges in designing a wireless sensor device are to “develop low power communication with low cost on-node processing and self-organising connectivity/protocols” and “another critical challenge is the need for extended temporal operation of the sensing node despite a (typically) limited power supply (and/or battery life)” [17].

There are a number of relatively low-cost hardware platforms that are used in Wireless Sensor Networks. These usually contain a micro-controller, such as one of the MSP ultra, low-power 430 family or an ARM Cortex-M3 processor, an RF chip, such as the Chipcon 2420, 2500 or 2650 and possibly some on-board sensors. These simple nodes often use low level operating systems like TinyOS [18] or Contiki [19] or TI's Simpliciti RF protocol stack [20]. In terms of their interface, sensor devices generally need support for the following operations:

- set their configuration (such as the thresholds at which to send alerts),
- get their data from local sensors and respond to get requests for it
- send alerts based on sensor events
- execute actions, e.g. in an actuator device or to reset sensor configuration

An interesting discussion and definition of classes of these devices is provided in [21], where the class indicates a device’s capabilities, as below:

- Class 0 has much less than 10 KiB of RAM for data and less than 100 KiB of Flash/ROM for code. Such devices are so constrained in memory and processing capabilities that while they may be able to communicate directly to the Internet, it will probably not be done securely and will require a more powerful gateway/proxy to support secure connectivity to the Internet. Examples include the original Tmote Sky [22] in Figure 2 and the TI SensorTag family of motes. The Tmote Sky only had 48KiB of ROM and 10KiB of RAM, allowing it to run a micro IP stack and CoAP, but not to support the cryptography code for security.

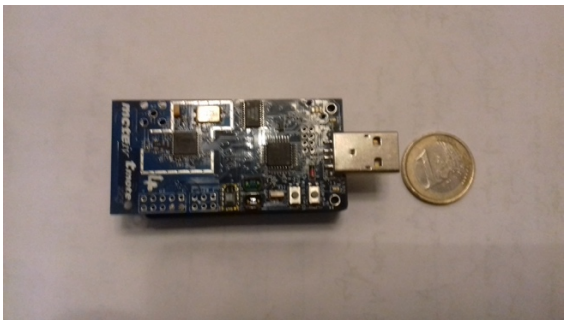

Figure 2 Tmote Sky Device [22]

- Class 1 has approximately 10 KiB of RAM for data and 100 KiB of Flash/ROM for code. They are able to use a protocol stack specifically designed for constrained nodes to limit memory use and power consumption, such as uIP on Contiki with the Constrained Application Protocol (CoAP) over UDP as an upper layer. As such they may not need a separate gateway. The Zolertia Z1 device [23] shown in Figure 3 is a typical node with a 16-bit RISC MSP430F2617 low power microcontroller with a 16MHz clock speed, 8KB RAM and a 92KB Flash memory using the CC2420 transceiver and is IEEE 802.15.4 compliant. It also comes with a built-in programmable accelerometer and a digital temperature sensor with ports for external sensors.

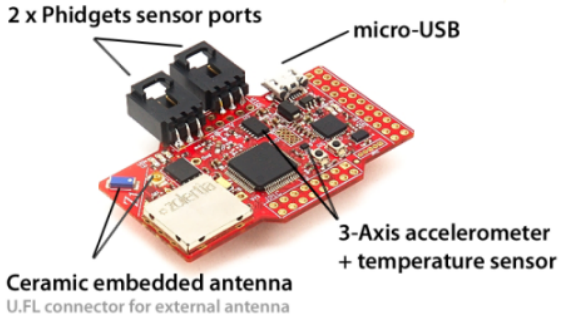

Figure 3 Zolerta Z1 Device [23]

- Class 2 has approximately 50 KiB of RAM for data and 250 KiB of Flash for code. Such devices are able to run, within limits (such as connections, routes), the same protocol stacks as general purpose servers, but the use of the stacks designed for more constrained devices would leave more resources such as memory for application code and data.

There are constrained devices with capabilities beyond those of Class 2, but still the environmental constraints such as power consumption and form factor must be considered. For example, SunSPOT [24] was a more powerful hardware platform able to run Java and while it reduced the difficulty associated with programming embedded network devices, it had limited lifetime and did not achieve sufficient use for continued development. Another example is the Raspberry Pi [25], which does provide a useful test and development environment as it is relatively low-cost and able to run Raspbian, which is a Debian-based Linux distribution. The Pi 2 Model B offers a 900MHz quad-core ARM Cortex-A7 CPU with 1GB RAM with USB, HDMI and GPIO ports (useful for attaching sensors). This processing capability allows the use of high level languages such as Python and applications such as the Mosquitto MQTT broker can be run on a Pi. Indeed, the newer Pi 3 Model B+ and Pi Zero W include Bluetooth Low Energy capability.

The types of sensor device found in a LoRa network could be considered as a class of device less capable than Class 0, as it is generally running a very simple application above a custom MAC layer to connect to a gateway device. LoRa is a spread spectrum modulation technique, which provides a long range and low power wireless network usually run by specific operators. One company (Semtech) provides the radio chips featuring LoRa Technology and a separate LoRa Alliance drives the standardisation of the LoRaWAN protocol [26]. This thesis is concerned with providing a holistic architecture for a seamless set of IoT services and so support for a proprietary model such as this would be provided by a custom component to integrate into the holistic architecture.

2.4 Wireless Sensor Network Stack

A wireless sensor network will use a communication stack consisting of a number of layers, with an example IP stack, including CoAP, shown in Figure 4. This section outlines some of the options for WSNs, particularly the wireless layers.

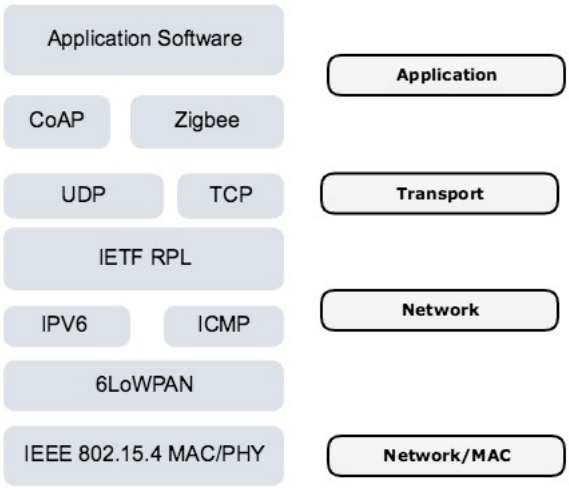

Figure 4 Network Stack including RPL

2.4.1 Wireless Interface Layers

Many WSN nodes use either IEEE802.15.4 or Bluetooth Low Energy for their Wireless interface. IEEE 802.15 wireless personal area technologies are designed for short-range, targeted applications, operating on unlicensed bands at lower power levels than the 802.11 standards. Bluetooth [27] (conforming to IEEE 802.15.1) operates at power levels from 1mW to 100mW and data rates of 1Mbps to 3Mbps using an Adaptive Frequency Hopping approach. Bluetooth Low Energy is designed for very low power operation, using a frequency-hopping spread spectrum approach supporting data rates from 125 Kbps to 2 Mbps.

Zigbee [28] evolved from IEEE Standard 802.15.4-2006 and is designed for low data-rate, low-power applications like sensor readings and interactive devices. Ultra-Wideband (UWB) [29] is a wireless technology proposed for several Body Area Network (BAN) and WSN scenarios, offering the ability to provide location information accurate to between 10 and 20cm. Narrowband IoT (NB-IoT) is a Low Power Wide Area Network (LPWAN) radio technology standard developed by 3GPP [30]. NB-IoT uses a subset of the LTE standard and is designed for indoor coverage, low cost, long battery life, and high connection density.

The evolution of 5G networks is increasingly becoming a driving force for IoT adoption and the design of platforms to utilise its emergence. 5G is expected to

have extended coverage, lower latency, higher throughput and bandwidth (of the order of Gbps in specific scenarios) and allow the deployment of tens of billions of devices (and sensors) over the Internet with improvements in the quality of service perceived by users [31].

Each of these wireless technologies has particular characteristics that determine their suitability for a given scenario. For example, Bluetooth would be suitable for a small-scale network requiring high data throughput and where connectivity to a standard mobile phone is desired. Zigbee might be chosen where lower data throughput is acceptable (up to 250Kbps) and lower power consumption is required. UWB would be chosen if its accuracy in determining location was required for an application. In particular, if a WSN is installed in a remote location, it must rely on battery power or energy harvesting such as solar to run for extended periods of months or years, making it necessary to use low-power radio chipsets and processors and implement power efficient Media Access Control schemes, where the device sleeps as long as possible.

A Body Area Network (BAN) is a short-range wireless network for consumer and healthcare applications. IEEE 802.15.6 defined MAC and physical layer standards for BANs and had an emphasis on ultra-low power consumption to extend the node's lifetime. It also addressed the issues caused by the challenging radio environment of the body area, aiming to provide a range of QoS support for a variety of applications and meeting form factor requirements for the wireless devices in wearable applications.

The open nature of the radio environment leads to the possibility of eavesdropping by malicious users and will require appropriate security algorithms to ensure data privacy, data integrity and protection against denial of service attacks. This need for security needs to be accommodated as a basic part of the communication protocols at the physical and data link layers, but also at the higher layers to determine if devices should interwork or collaborate at all. Four basic security requirements can be stated as Confidentiality (guarantee the contents of the message are disclosed only to authorized individuals/endpoints), Authorization (guarantee the sender is authorized to send a message), Data integrity (guarantee the message was not modified in transit, by accidental or deliberate means),

Refutability (guarantee the message was sent by a properly identified sender and is not a replay of an earlier message).

2.4.2 Network and Routing Layers

WSNs are limited by the short radio range between nodes and hence they are often deployed in a structured manner using star or clustered topologies with appropriate routing strategies. In such cases, nodes are arranged around a coordinator a single hop away (as in Zigbee) with those co-ordinators connected among themselves or the nodes use multi-hop routes with each node routing requests/responses. Alternatively, some WSNs are deployed in an ad-hoc manner with sensors scattered over an area of interest such as a battlefield area. These deployed sensors then self-organise to form a network establishing hierarchies, clusters, routing strategies, radio channels and would be expected to work for a period of time without human intervention, probably until the battery powering the sensor device is exhausted [32].

Whether deployed in a structured or ad-hoc manner, the wireless interface is responsible for providing:

- discovery of wireless devices in the surrounding environment.
- establishment of appropriate (e.g. sufficiently reliable) communication links with all or a subset of sensors in the network.

Other layers in the communication stack must then provide (possibly in a way specific to a given wireless interface)

- routing as required to forward information to/from the sensors.
- establishing connection(s) to external network(s), if a node is a gateway.

User applications usually communicate to a sensor network via a dedicated node or nodes that handle user requests for information and forward the responses to them. This sink node may act as a gateway to another network, such as a mobile network and is generally more powerful, in terms of processing and memory, than the sensor nodes themselves. Furthermore, such a gateway node may be mains rather than battery powered or be able to be recharged easily, so lessening the electrical power constraints that the battery powered sensor nodes operate under.

A smartphone is a suitable platform in terms of processing and network connectivity for such a sink, particularly in consumer applications. This centralised node has full knowledge/control of its network and responsibility for establishing and maintaining the network to the sensor nodes, including MAC level functionality. This sink node also provides the WSN with an interface to the external mobile network, including appropriate security support such as access control. The sink is, however, a potential single point of failure as may happen when it is out of range of the sensing node(s). Other scenarios, such as high-performance links from a static WSN to the Internet or some medical applications would require a dedicated gateway to an external wireless network.

The use of IP in constrained devices provides scalability and simplifies the development and deployment of both applications and networks by the use of existing protocols and tools. Contiki provides the open-source uIP stack which requires less than 5KB of code space and a few hundred bytes of RAM. The use of IP is complemented by initiatives such as the IP Smart Objects (IPSO) Alliance which uses IPv6 LoWPAN to introduce an adaptation layer to enable efficient (i.e. less power consuming) IPv6 communication over IEEE 802.15.4 LoWPAN links [33]. The IETF Routing Over Low power and Lossy network (ROLL) working group also defines routing solutions for Low power and Lossy Networks (LLNs), specifically stating that “LLNs are transitioning to an end-to-end IP-based solution to avoid the problem of non-interoperable networks interconnected by protocol translation gateways and proxies” [34]. In particular, the IETF has defined RPL as the IPv6 routing protocol for low power and lossy networks.

2.4.3 Wireless Sensor Network Routing

WSNs may be broadly classified as

- static with sensor nodes that are not mobile and/or join/leave infrequently, e.g. smart grid systems.
- dynamic in nature where nodes are mobile and/or join/leave frequently, providing information in possibly different contexts, locations and as members of different WSNs that may collaborate.

Routing can be categorised similarly into static, dynamic or hybrid:

- static routing uses fixed paths to route packets to their desired destination, regardless of the state of the network. IP networks select the destination on a hop-by-hop basis using a statically-constructed routing table, e.g. using utilities such as iptables, to forward the packet, rather than computing the entire path in advance. For end user devices this usually consists of simply forwarding packets to the gateway of that particular network as determined by its IP address. The main advantage of static routing is its simplicity. As such, it is appropriate for small networks or for larger networks with defined hierarchies and subnets (each with assigned address ranges and delegated authority for address assignment). Maintenance of these routing tables becomes difficult as the (sub)networks get larger or more dynamic particularly in terms of interconnection with other networks.
- dynamic routing reacts to changes in the status of the network and removes/reduces the role of the system administrator in route maintenance by using routing protocols to exchange routing information. Each router sends the destinations it can reach to its neighbours, which can then update their own routing tables accordingly. In this way, dynamic routing should make larger networks easier to configure and maintain, but also more adaptable to topological changes such as when nodes/links fail or configuration changes occur. These changes are more likely to occur in wireless networks where nodes are mobile and/or where there are possibly frequent changes in the network topology. The improved scaling properties of dynamic routing are, however, offset by the increased implementation complexity and the increased (routing) traffic as each node in the network routes data packets to/for other nodes. In the context of WSNs, the increased complexity of dynamic routing makes demands on the node hardware (and software) and the routing traffic must be minimised to reduce power consumption.
- hybrid routing uses both static and dynamic routing schemes in different parts of the network, e.g. static routing in the access network where the topology changes are more limited and dynamic routing in the core network, allowing the use of cheaper/simpler devices in the access network and reducing the routing traffic overhead as network capacity is more likely to be limited there than in the core network (and this is especially true if the access network is wireless).

A Wireless Sensor Network specific categorisation of routing is as follows [17]:

- Data-centric – this is based on combining the data from multiple, different sources en-route to the eventual destination and contrasts with the normal address-centric approach of sending data between pairs of endpoints. WSNs using data centric approaches may not have unique identifiers for each node, making direct queries problematic and requiring a data centric, aggregated approach, e.g. directed diffusion.
- Hierarchical - this overcomes the issues of a single centralised gateway by using layers of clusters, with a cluster head for each and using multi-hop communication within a cluster and possibly aggregating at the cluster head. Examples include Low Energy Adaptive Clustering Hierarchy (LEACH) [35] and Power Efficient Gathering in Sensor Information Systems (PEGASIS) [36].
- Location-based – location information (such as from GPS) is used to calculate the distance between two nodes and so determine/estimate the energy consumption in routing. Greedy Perimeter Stateless Routing (GPSR) [37] is based on forwarding packets based on location to nodes that get the packets closer to the eventual destination. It is greedy as the node always forwards to the node (within its radio range) that is closest to the destination. Geographic Hash Tables (GHTs) [38] allow data queries to be sent to the node storing the named data, but can potentially hash outside the geographic boundaries.
- QoS-oriented – these consider the end-to-end delay requirements in setting up paths in the sensor network. Examples include Stateless Protocol for End-End Delay (SPEED) [39].

RPL is the IPv6 Routing Protocol for Low Power and Lossy Networks. The design of RPL used defined routing requirements from the application areas of home automation [40], building automation [41], urban networks [42] and industrial networks [43]. Even in the home scenario, the heterogeneity and number of devices is noteworthy with devices ranging from resource constrained smoke alarms to home health monitoring devices for blood pressure to video capture devices and from tens of devices to hundreds in an industrial setting to tens of thousands in an urban setting. This resulted in RPL supporting multipoint-to-point, point-to-multipoint and point-to-point, where multipoint-to-point is for

the typical monitoring use case of reporting sensor readings to a central point. Point-to-multipoint is more used for sending control commands for actuators or for processing queries. Point-to-Point is more used where devices can share local information and react accordingly, e.g. nearby machines in a closed loop control system [43]. The scenarios driving these requirements showed devices being mostly fixed, with limited mobility as in mobile workers or moving vehicles [43], but the requirement was that the moving device re-established communications with a static device.

RPL is a proactive distance vector protocol, which builds Destination Oriented Directed Acyclic Graphs (DODAGs) rooted towards one sink node or Local Border Router (LBR), which has a unique identifier. Each node is given a rank to determine its relative position and distance to the LBR (higher rank for nodes further away) in the DODAG. The DODAGs are optimized using an Objective Function using metrics such as hop count, latency, expected transmission count and energy used. RPL uses the Trickle algorithm [44] to maintain consistency between neighbours due to topology changes while also reducing the transmits to achieve this. A network may have several independent RPL instances and an RPL node may belong to more than one RPL instance and also act as a router.

A specific recognition of the constrained resources in WSN nodes in RPL is the provision of storing and non-storing modes of operation, where storing mode means a node keeps a routing table. In non-storing mode only the root node holds a routing table, meaning that for nodes to communicate the packet has to go via the root, whereas it can go via a common node higher in the graph for storing mode. It is important to note that RPL does not currently allow both modes to operate in the same network.

Although the acknowledged de-facto routing standard, an article in 2016 considered RPL in the light of experience from implementations and emerging IoT application requirements [45]. For example, it cites the need for work being carried out on “Reactive Discovery of Point-to-Point Routes in Low-Power and Lossy Networks (P2P-RPL)” in RFC 6697, due to the overheads, latency and congestion by having point to point communications routed via a node higher in the graph. It also highlights the limits of mobile devices having to re-establish communication with a static device resulting in obsolete routing information and

packet loss. It also points out that the use of IPv6 addresses (and compression in 6LoWPAN) combined with not being able to run both storing and non-storing modes restricts its use in very limited nodes, e.g. the routing table was limited to 50 entries on a TMote Sky with 10KB of RAM and the code size of RPL is almost double that of AODV. Furthermore, this limited scalability due to having to store routes (particularly for nodes close to the root node) and maintain links in a changing radio environment (due to static Trickle thresholds) adversely affects the reliability of the downward paths and point to point communications as the number of nodes increases and the table is held to a set size (e.g. 50). This is particularly important as [45] points out that point-to-multipoint and point-point communications in scenarios involving mobile devices are the emerging use cases as IoT applications develop. Based on this, this article suggests that a single routing standard is unlikely to be able to handle such a range of device heterogeneity and application requirements, meaning that RPL should become a framework able to include specific applications and device requirements in an interoperable manner.

2.4.4 Wireless Sensor Network Application Layer Protocols

2.4.4.1 Constrained Application Protocol (CoAP)

The micro IP stacks developed for constrained WSN nodes combined with IPv6 over low power wireless (6LowPAN) [46] have enabled the development of application level protocols such as the Constrained Application Protocol (CoAP). CoAP has been developed by the Internet Engineering Task Force (IETF) and is targeted at the IoT area [5]. It is a standard for a specialised web transfer protocol for constrained nodes and constrained (e.g. low-power, lossy) networks. It is built on top of UDP and uses a small, simple header of less than 10 bytes, including a 16 bit message identifier. It also uses a UDP binding with reliability, using ACKs, and multicast support, although recent work has extended it to TCP, TLS and WebSockets in RFC8323 [47]. It uses web concepts such as URI's and media formats for easy integration of such constrained environments into HTTP. It addresses issues such as the overhead of HTTP headers and TCP performance over lossy links and the handling of sensor node duty cycles. It uses the REST architectural style [48], where resources (such as sensors) are represented in a number of formats using a subset of the IANA Internet media types and accessed

by their Universal Resource Identifier (URI) using its own schema, coap://. It has a limited set of verbs, such as GET, POST, PUT, DELETE in HTTP, as it is designed to be easy to proxy to/from HTTP. Unlike HTTP, CoAP is a binary format. The RESTFul based style facilitates application development and scalability as a result of its decoupled nature.

CoAP also provides Resource discovery via the Resource Directory (RD) and specific message types such as Confirmable (CON) to provide reliability. The use of an “observe” flag in the CoAP GET Request provides observe/notify on a given resource, effectively offering a Publish/Subscribe model.

CoAP has been implemented on Contiki [49] and this implementation is also used as the basis for the implementation work on Contiki presented in later sections. An example IP and RESTFul approach with CoAP is an end-to-end IP based architecture for greenhouse monitoring by integrating CoAP over a 6LowPAN WSN using Contiki [50].

2.4.4.2 Message Queue Telemetry Transport (MQTT)

The Message Queuing Telemetry Transport (MQTT) [51] is a Machine to Machine (M2M) asynchronous protocol that is an alternative to Web-like protocols. It is an open standard from the OASIS consortium. It was designed to be bandwidth-efficient and use little battery power. It uses binary messages to exchange information with low overhead, but unlike the original CoAP it uses TCP, or other network protocols that provide ordered, lossless, bi-directional connections, for transmission. It is designed to be easy to implement and requires a small code footprint, e.g. being able to run on a controller with 256KB of RAM.

It is based on a publish/subscribe approach that allows messages from devices to be sent (published) to interested (subscribed) services in contrast to the HTTP request/response paradigm. MQTT uses a central broker, where messages are published to a broker on a topic and the broker filters messages based on topic and distributes messages to subscribers for that topic. There is no direct TCP connection between a publisher and subscriber. Publish and subscribe packets contain a PacketId that is unique between client and broker. It also provides three qualities of service for message delivery; "At most once", "At least once" and "Exactly once”.

Use of the publish/subscribe message pattern provides one-to-many message distribution and a degree of decoupling of applications from the sources/sinks of data, although its simplicity and lack of prescription means it relies on publishers/subscribers to agree on topics (including uniqueness). MQTT clients must be configured with a dedicated broker service and this tight coupling limits extensibility and limits its adaptability to an evolving environment [10]. MQTT is used in occasional dial-up connections with healthcare providers and in a range of home automation and small device scenarios [51]. It is also used as part of Amazon's IoT Services.

2.5 Wireless Sensor Network Information Models

An important aspect to be considered in a WSN is how to store and represent the variety of data on often constrained devices so that application software can understand data from sensors and actuators in the way people using browsers understand information on the Web [52]. This section considers information models that have been used for sensors.

2.5.1 IPSO and OMA LWM2M

IP for Smart Objects (IPSO) is an alliance of interested parties. It promotes and documents the use of IP-based technologies, defined by standard organizations such as IETF, for smart objects (such as sensors for light, pressure, temperature, vibration, actuators and other similar objects) and to support a range of interoperation use cases [6]. IPSO objects can be used to encapsulate sensor data, links and metadata and be accessed using a URI. IPSO objects do not require the use of CoAP and the Open Mobile Alliance Lightweight Specification (OMA LWM2M) [53] is used to develop interoperable solutions. The current IPSO Smart Object definitions are mostly for sensors/actuators rather than the broader range of smart objects envisaged in [54], where smart objects can act on their own and exchange information with humans, with an agent based middleware layer supported by Cloud services to create sets of cooperative smart objects.

OMA Lightweight Machine to Machine (LWM2M) uses a RESTFul approach with CoAP. In an IoT context it swaps "server" and "client" roles so that a node runs at least a CoAP Server and a LWM2M Client, rather than being simply a

client. This requirement to act as a server may be limiting in some circumstances, e.g. where a security policy only allows outgoing connections.

LWM2M provides a simple and reusable object model with a set of interfaces for managing constrained devices, which includes Bootstrap, Registration, Information Reporting, Device Management and Service Enablement. IPSO Smart Objects are extensible objects based on the LWM2M data model, which includes objects for a range of entities, including basic sensors and actuators. These basic objects are represented using a simple common data model and resource template. The model consists of Object Instances, Resources (instances) with reusable resource and object identifiers combined into a URI to identify a resource, e.g. the URI 3303/0/5700 represents a "Sensor Value" (resource identifier 5700) in a "Temperature Sensor" (object identifier 3303) instance (id of 0). More complex objects can be composed to represent items that contain multiple resources, e.g. an IPSO Thermostat (8300) may have IPSO temperature sensors, (3303), IPSO Setpoint (3308) and IPSO Actuation (3306) [55].

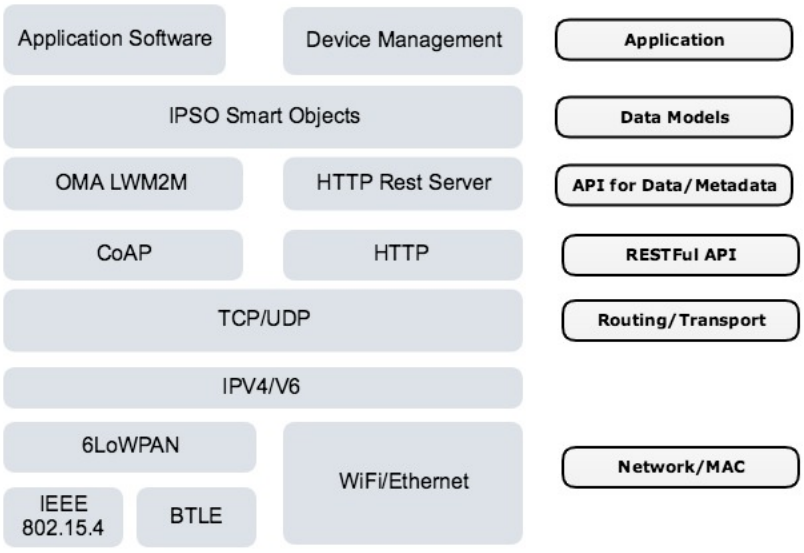

Figure 5 OMA LWM2M and IPSO

Figure 5 shows how IPSO Smart Objects, LWM2M, CoAP and 6LOWPAN form a protocol stack that can provide end to end interoperability between constrained devices and services.

These data models have the potential to give greater application interoperability and to ease the difficulties imposed by the heterogeneous nature, limited development environments and interfaces of existing solutions if adopted sufficiently widely. A general point regarding the use of IP as promoted by IPSO is that it not only enables integration with the Internet and IoT, but also with the Web of Things (WoT) and the REST architectural style [56]. Furthermore, the use of IPv6 makes it possible to provide IP addresses for individual devices.

2.5.2 Common Information Model (CIM)

Although not specific to WSNs, the Common Information Model (CIM) [57] is a rich information model that does include sensors. It is defined by the Desktop Management Taskforce (DMTF) as the management information for systems, networks, applications and services. Its schema (version 2.49, released in 2017) provides a comprehensive object oriented model for managing the components found in computer systems. It defines a CIM_Sensor object with the following generic attributes/properties, using qualifiers for these attributes to cater for a specific sensor.

- SensorType – example values are Temperature, Voltage, Current, Tachometer, Counter, Switch, Lock, Humidity, Other.
- OtherSensorTypeDescription - used when the SensorType property is set to "Other"
- PossibleStates - specific to a sensor type, e.g. a NumericSensor supporting thresholds can report states such as "Normal", "Upper Fatal", "Lower Non-Critical".
- CurrentState - must be one of the PossibleStates
- PollingInterval

The CIM_Sensor also inherits from the following higher layer objects:

- CIM_Logical_Device - for identifying and descriptive information and information related to how long it has been powered on
- CIM_EnabledLogicalElement – for information related to state changes
- CIM_ManagedSystemElement - for current status information, such as "OperationalStatus "
- CIM_ManagedElement - for instance identifier and name

A CIM_Sensor also inherits a number of methods from its superclasses, such as Reset, SaveProperties, RestoreProperties RequestStateChange. The subclass CIM_NumericSensor (returns numeric readings and optionally supports thresholds settings) was designed to replace a range of specific sensor classes such as CIM_CurrentSensor, CIM_TemperatureSensor, CIM_VoltageSensor by setting the appropriate SensorType property from CIM_Sensor.

Data defined using CIM can be accessed in a protocol independent manner using CIM-XML, which can operate over protocols such as HTTP. The basic operations supported are for Read/Write on properties, classes and instances (e.g. GetProperty, SetProperty , GetClass, GetInstance, EnumerateInstances, EnumerateClass), Instance Manipulation (e.g. DeleteInstance, CreateInstance), Schema Manipulation (e.g DeleteClass), Association traversal (Associators, References) and Methods defined in specific classes.

The use of XML and the string based nature of many of the attributes and of CIM-XML itself make it too verbose for use in the constrained WSN environment, but its definitions align well with the earlier description of sensors in WSNs. Thus, the basic functionality required by a management application is the same in a WSN as in a more traditional sensor application. Hence, the emphasis should not be on redefining the use of sensors or the sensors themselves, but in doing so as efficiently as possible given the WSN constraints and in a way which can map easily to formats used by the higher level applications.

2.5.3 SensorML

Unlike OMA LWM2M and CIM which define the sensor and its attributes including the actual readings, SensorML provides models and an XML encoding for describing a process, particularly the process of measurement by sensors and instructions for deriving higher level information. Processes define their inputs, outputs, parameters and method, as well as providing relevant metadata. Header information defines the schema and namespaces, an identifier contains a unique identifier (a UUID, URN, URL, or simple text) for any service or resource associated with this sensor and a sensor description saying “what it measures” and "where it is". [58]. It does not encode measurements taken by sensors; measurements can be represented in TransducerML. SensorML provides the ability to describe a sensor (or other online processing component) and to provide a link to the real-time values from it.

2.6 Web of Things (WoT)

The Web of Things (WoT) architecture is presented in [59] as a W3C candidate recommendation. It uses the concept of Web things, which are used by consumers. A thing is the abstraction of a physical or virtual entity (e.g. a device) and the W3C WoT specifies that its metadata must be described by a standardised WoT Thing Description (TD) to provide the external representation of a thing. A TD describes an individual thing's functions and interfaces, including information models, transport protocol description and security information. The TD is based on the JSON representation format and is machine-understandable.

Consumers must be able to parse and process the TD format. It is designed to allow consumers to discover and interpret the capabilities of a thing (through semantic annotations) and adapt to different implementations when interacting with a thing, providing interoperability across different IoT platforms and standards. This recommendation identifies the following building blocks to improve the interoperability and usability of IoT:

- Web of things (WoT) Thing Description [60]
- Web of things (WoT) Binding Templates [61]
- Web of things (WoT) Scripting API [62]
- Web of things (WoT) Security and Privacy Considerations [63]

This recommendation [59] also describes a set of use cases with common patterns and application domains (similar to those described in section 2.7) and a set of requirements for WoT implementations. The requirements are broken into common principles and a subsequent set of functional requirements. The common principles are as follows:

- “WoT architecture should enable mutual interworking of different eco-systems using web technology.”
- “WoT architecture should be based on the web architecture using RESTful APIs.”
- “WoT architecture should allow to use multiple payload formats which are commonly used in the web.”
- “WoT architecture must enable different device architectures and must not force a client or server implementation of system components.”

- “WoT architecture should be able to be mapped to and cover all of the variations of physical device configurations for WoT implementations.”
- “WoT should provide a bridge between existing and developing IoT solutions and Web technology based on WoT concepts. The WoT should be upwards compatible with existing IoT solutions and current standards.”
- “WoT must be able to scale for IoT solutions that incorporate thousands to millions of devices.”
- “WoT must provide interoperability across device and cloud manufacturers. It must be possible to take a WoT enabled device and connect it with a cloud service from different manufacturers out of the box”

The functional requirements provide more granularity in the areas of thing functionalities, search and discovery, description mechanism, description of attributes, description of functionalities, network, deployment, application and legacy adoption. For example, the search and discovery functional requirement specifies that

- “WoT architecture should allow clients to know thing's attributes, functionalities and their access points, prior to access to the thing itself.”
- “WoT architecture should allow clients to search things by its attributes and functionalities.”
- “WoT architecture should allow semantic search of things providing required functionalities based on a unified vocabulary, regardless of naming of the functionalities.”

It also specifies that a thing’s functionality requirements are:

- “WoT architecture should allow things to have functionalities such as reading thing's status information”
- “updating thing's status information which might cause actuation”
- “subscribing to, receiving and unsubscribing to notifications of changes of the thing's status information.”
- “invoking functions with input and output parameters which would cause certain actuation or calculation.”

- "subscribing to, receiving and unsubscribing to event notifications that are more general than just reports of state transitions."

It is worth pointing out in the context of devices that WoT is very specific in stating that "information models defines device attributes, and represent device's internal settings, control functionality and notification functionality. Devices that have the same functionality have the same information model regardless of the transport protocols used."

The architectural approach of WoT uses the concept of interaction affordances to make the TD metadata self-descriptive. An affordance is an abstract model of consumer interaction with the thing and is not particular to a given network protocol or data encoding. This allows consumers to identify the capabilities of a thing and how to use those capabilities.

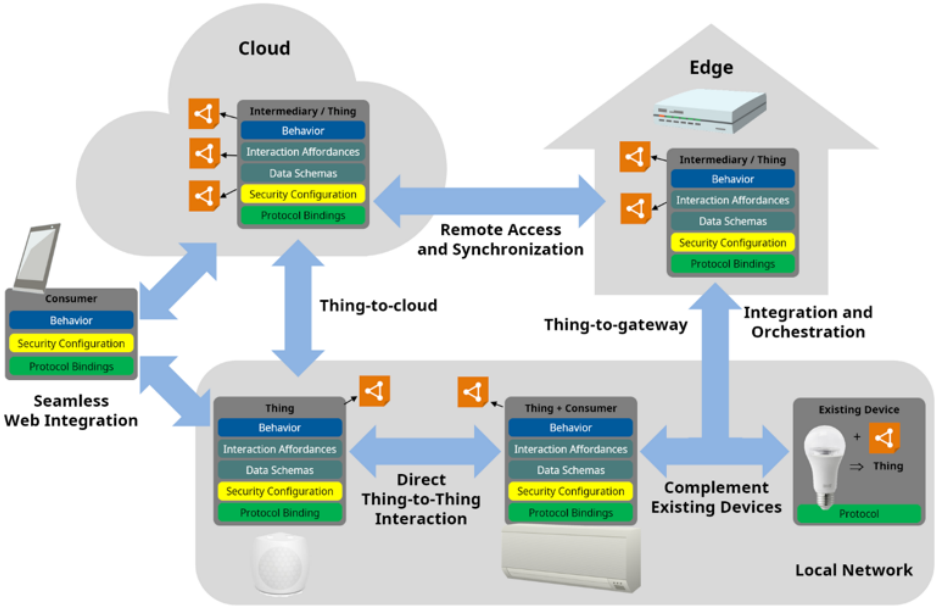

Figure 6 Abstract Architecture of W3C WoT [59]

A web thing has the following architectural aspects shown in Figure 6:

- interaction affordance, which can consist of the thing's properties, actions and events. A Property exposes the thing's state. An action invokes a function of the thing and an event asynchronously pushes state transitions from a thing to a consumer.

- behaviour, possibly autonomous, and the handlers for the interaction affordances
- security configuration to control access to the interaction affordances and to manage related metadata
- protocol bindings map the interaction affordances to messages within a protocol.

A servient is a term used for a software stack that implements the WoT building blocks. It may host/expose things and/or host consumers that consume things, possibly supporting multiple protocol bindings to allow different IoT platforms to interact with each other.

The recommendation [59] shows the following deployment scenarios for WoT, which can also be seen in the abstract architecture in Figure 6:

- consumer and thing on the same network. This is the simplest case, where an intermediary is not required.
- consumer and thing connected via intermediaries, which may be proxies or digital twins. The intermediary plays both thing and consumer roles. A proxy intermediary must get a TD of a thing and creates a consumed thing for the exposed thing it communicates with. It must then create a proxy object of the thing as a software implementation with a TD for the proxy object. This proxy object may have a new identifier and new communications metadata. The final step is for the intermediary to create its own exposed thing for the original thing, which consumers can communicate with (possibly using a different protocol to the actual thing).
- discovery using a thing directory. The directory holds the TDs of local devices implemented as things that registered with a thing directory service. A client application can query this directory to obtain the metadata to contact local devices.
- service-to-service connections across multiple domains. This allows cloud eco-systems based on different IoT platforms to be interoperable and create a larger system-of-systems, where this is done by either synchronising the thing directories or synchronizing proxies.

2.7 Characteristics of IoT Applications using Wireless Access

There are a wide range of scenarios in which wireless access via mobile networks (and 5G in the future) and/or Wireless Sensor Networks are used, usually where the limits of wireless technologies such as limited range or device lifetime are acceptable [64], particularly given the flexibility offered by the wireless connection to be deployed close to physical phenomena and not require cabling. Such scenarios include those of military (e.g. situation awareness, battlefield sensing), emergency (e.g. disaster management, hazardous chemical monitoring), environmental (e.g. soil, water, habitat, monitoring), medical (e.g. respiratory rate, oxygen measurement), industrial (e.g. equipment monitoring), home (meter reading, appliances), automotive (e.g. Tyre pressure monitors) [17].

Traditionally, applications using wireless access (and sensor networks in general) are concerned with “dumb” data collection, where the sensor network is treated as a peripheral network to provide data to external domains such as the Internet or LANs. Specific topologies, such as star, with limited, possibly fixed, numbers of nodes may be used to make configuration simpler in scenarios such as environmental monitoring in a fixed geographic area, infrastructure monitoring or monitoring temperature or light in a house. These scenarios are characterised by fixed types and fixed numbers of sensors, short range and limited interworking.

More advanced scenarios exist and are characterised by a greater number of (possibly heterogenous) nodes or nodes that enter/leave the wireless network or networks that have to interwork and collaborate. This is particularly true in the IoT and 5G scenarios, which envisage that sensor networks will increasingly feature a greater variety of applications in some of the areas above, i.e. urban sensing, smart homes, smart cities, body sensor networks and healthcare, home automation and industrial networks. An example of such an advanced scenario is where first responders enter a building and want to get information from the building's WSN and use the mobile network to collaborate or even where firefighters deploy the nodes when they enter the building [65]. By taking advantage of the potential of IoT, such applications challenge many of the simplifying assumptions currently made for WSNs, as these applications will have

wider deployment, greater scale, heterogeneity of sensors, high data rate and multiple users.

These applications can be classified on the basis of the network characteristics they require and how they relate to the Quality of Service, as has been done for medical applications by IEEE 11073 [66]. IEEE 802.15.6 broadened these requirements as part of its BAN studies and examined a range of applications [64] under the categories of safety, radio regulation, topology, data link speed symmetric/asymmetric, data rate, number of devices, duty cycle, range, co-existence, robustness/reliability, power consumption, the possibility of energy scavenging, error sensitivity, latency sensitivity, setup time, location awareness, form factor, privacy, power source, cost, standards compliance required. It also divided BAN applications into the broad categories of medical and non-medical. The medical area can be divided into wearable BAN (e.g. EEG, ECG, Temperature), implant BAN (e.g. drug delivery) and remote control of devices (e.g. insulin pump). The non-medical applications can be broken into real-time audio streaming, real-time video streaming, file transfer, stream transfer and entertainment (gaming, social networking) [64]. As an example of the potential benefits of IoT, its use with BAN wireless technology leads to the possibility of widespread untethered medical and health monitoring without the need for the use of cables for such systems. This will provide greater flexibility in placing equipment, allowing the collection of patient data to no longer be limited to the bedside or wired points so enabling patient mobility in hospital and home monitoring. These benefits would allow earlier release from hospital and the development of new monitoring and alerting applications, subject to deploying with appropriate simplicity, security and reliability.

The approaches above are examples of mapping application requirements to specific network parameters, but it is also useful to consider the nature of the applications that use the data delivered by sensors and WSNs:

- Data Flow –sensed data is usually sent to a single sink/application and that data is usually time stamped as it is time dependent (to a degree determined by the entity being sensed and/or the application's requirements). Data transmission can be initiated by the sensor (it reports on a periodic basis to a sink or sends an alert) or will be sent as a result of

an application's action, such as a query/request or a command to the device (e.g. to set thresholds, control an actuator).

- Local Data Processing - depending on the application scenario (e.g. the extent of geographic locality of devices, time limits on reporting of data) and the node capabilities, data may be processed locally on the node so that it is aggregated, encrypted on the sensor node before being transmitted or simply stored for query by the application. An application should be aware of the capabilities of the device in this regard.
- Data Reporting – this may be performed at a fixed rate or may vary as the sensed phenomena changes, e.g. more frequent readings may be required in an emergency situation, or it may vary depending on the requirements of different applications or it may be limited by the ability of the sensor itself. Depending on the nature of the data reporting, it may be possible for the application to schedule data reads (or sends by the sensors) as part of configuration or agreed as part of initialisation (by configuration or negotiation). Note the data received by the application does not have to be data from a single actual sensor, i.e. it could be aggregated data.
- Event Generation - sensing can result in unexpected events such as alerts on thresholds being exceeded that need to be handled and reported to the appropriate sink/application. There may be a latency requirement on handling such events, which will impose requirements on lower layers to ensure that all nodes on a route to deliver that event can be woken up or are scheduled to wake up in time.
- Data Priority – some applications may require that the data they generate is treated at a higher priority than other data and require special treatment at the lower layers of the protocol stack, e.g. alarms from medical devices.
- Energy – the application scenario may impose particular lifetime requirements that must be met, e.g. a node must last for a certain period before requiring its batteries to be replaced such as in habitat monitoring applications.

These higher layer characteristics of applications in IoT influence directly the nature of the network and architecture, e.g. the volume of data and need for local data processing are reasons for the emergence of fog and edge computing. In terms of WSNs specifically, these characteristics affect the network, data link and

physical layers, e.g. the number of nodes may rule out certain topologies (such as STAR for large numbers) or if the sensor readings are regular and can be scheduled across the sensor devices in a given network then a TDMA based MAC scheme is appropriate. On the other hand, if they are less predictable such reservation of time slots may be wasteful of power (or introduce latencies for waiting nodes) and a contention-based scheme may be more appropriate. Similarly, the use of a heartbeat (as in a generic wireless protocol such as Zigbee) to keep a connection open and/or its availability known is appropriate where data can be sent at any time. It may not, however, be required in certain WSN scenarios where the data is sent infrequently and the power used in sending/receiving a heartbeat would be wasteful, although it may be required to counter clock drift between nodes and ensure time synchronisation.

2.8 Summary

This chapter has given an overview of IoT and the role of Wireless Sensor Networks, their device hardware, software and network stack, including routing and application layer protocols. The developments in device hardware and supporting software have resulted in the deployment of small scale WSNs with relatively few sensors connected in a star or clustered topology with a gateway node to connect to the Internet or a mobile network. In these scenarios, the types and number of sensors are fixed, the range is short and interworking between WSNs is limited. It is expected that more of these applications will be deployed with the increasing availability of the software and hardware, together with reductions in cost, although limitations in interoperability and difficulties in predicting suitable sensor location for radio coverage are proving to be limiting factors in deploying WSNs. More advanced application scenarios are emerging in IoT and an overview of their characteristics was given. An overview of current technologies such as CoAP, MQTT and WoT was given, which will be considered further in following chapters in order to develop the holistic architecture.

This chapter has also considered defined information models for sensors and IoT, such as IPSO Smart Objects and the Open Mobile Alliance Lightweight Specification (OMA LWM2M) and also a more general purpose information model (CIM). These models showed that a node's sensor capabilities can be

defined as they are determined by the nature of the sensor and are not unbounded (as in other network endpoints). Furthermore this definition of capabilities could be advertised to other nodes and applications for their use in working with sensors in the network or dynamically handling new sensors that are added to the network. The IPSO and LWM2M models have the potential to provide greater application interoperability for services using this sensor data than point solutions using proprietary data models. As such, the ability to support the implementation of these models will be an important test for the proposed holistic architecture in terms of the code size required and the quality of abstractions provided to the application programmer to add and use those information models.

It can also be assumed that battery technology will improve and that lower power processors and radio components will be developed. Without corresponding work on approaches to match the application, network dynamics, routing and sensor node parameters it is likely that these technology gains will be exploited in specific point solutions for those limited scenarios outlined above.

Furthermore, if large numbers of WSNs or BANs develop as envisaged, there will be more sensors and actuators than there are Internet hosts currently. In this scenario, treating these networks of sensors (or other constrained devices) as peripheral devices and connecting them to the Internet via proxies or sinks will limit performance and scalability [67].

In summary, it can be said that the requirements for a WSN consist of ensuring adequate device lifetime, providing ease of software development, supporting autonomous operation and providing ease of deployment, especially on heterogenous nodes. These nodes have limited device processing and storage capability and limited (or proprietary) development environments and interfaces. These requirements provide significant software development challenges. The next chapter will consider architectural and technology approaches that may be able to address these challenges and following chapters will consider these in the specific context of WSNs and present an analysis resulting in a holistic architecture and implementation.

3 Distributed System Concepts

3.1 Introduction

Earlier chapters have shown that IoT and WSNs can be considered as distributed systems comprising WSNs, gateways, proxy devices and services with heterogenous device, networking and software technologies. This chapter considers a number of approaches in distributed computing and their current use to handle the heterogeneity and scale of IoT and WSN systems and data. It considers the RESTFul architectural style given its success in terms of adoption and scale in HTTP and web applications. It considers P2P systems from the viewpoint of their robustness, scalability and implementation complexity. It considers cache algorithms in terms of their effectiveness and implementation complexity. It outlines tuple-spaces as a possible basis for decoupling elements of the system and providing a simple API for developers. This consideration of these distributed system approaches has the goal of determining if these benefits can be realised in a holistic architecture for WSNs and other IoT scenarios, which will be analysed in Chapter 4.

3.2 Architectural Approaches

3.2.1 RESTFul Architecture Style

The Constrained Application Protocol (CoAP) was outlined in section 2.4.4.1 as an example of a RESTFul approach. The RESTFul architectural style uses a *resource* as a key abstraction of information that can be represented in a number of representations using the Internet media types. It is based on five interface constraints as follows [48]:

- All important resources are identified by a single resource identifier mechanism, usually a Universal Resource Identifier (URI). This constraint leads to the interface being simple, visible, and reusable.

- Access methods have the same semantics for all resources. For HTTP, this results in a limited set of verbs, such as HEAD, GET, POST, PUT, DELETE with well understood semantics. This constraint leads to the interface being visible, scalable, and available (by enabling the use of layered system, cacheable, and shared caches).
- Resources are manipulated through the exchange of representations. This constraint leads to the interface being simple, visible, reusable, cacheable, and evolvable using information hiding.
- Representations are exchanged via self-descriptive messages. This leads to the interface being visible, scalable, available and evolvable.
- Hypertext as the engine of application state (HATEOAS). This leads to the interface being simple, visible, reusable, and cacheable through data-oriented integration, evolvable via loose coupling, and adaptable though late binding of application transitions.

The RESTFul architectural style also includes processing elements that are determined by their roles in an overall application action, i.e. origin server (e.g. Apache), gateway proxy, user agent (e.g. Web browser). A recent paper reflecting on the RESTFul architectural style [68], including the original authors, considers that there have been different interpretations of the term REST, but reiterates that *"REST is not an architecture, but rather an architectural style. It is a set of constraints that, when adhered to, will induce a set of properties; most of those properties are believed to be beneficial for decentralized, network-based applications, while others are the negative trade-offs that can result from any design choice*". Importantly it also states that "*REST does not directly constrain the Web's architecture. Rather, an application developer may choose to constrain an architecture in accordance with the REST style*". The RESTFul style has been shown to facilitate application development and scalability as a result of its decoupled nature.

3.2.2 Middleware Approaches

One middleware approach is to provide separate components and abstractions for different parts of the overall system as the functionality required increases and the hardware becomes more capable. A good example of this approach is the Eclipse IoT architecture, shown in Figure 7 as three distinct software stacks [69]. The first

stack is for constrained devices, containing OS, Hardware Abstraction and Communication layers, with remote management across layers. The second stack is for gateways, which aggregate data and coordinate the connectivity of these devices to each other and to an external network. It includes layers to support IoT protocols, network management and data management/messaging. It will run on an OS with more functionality and may provide containers or specific application environments, e.g. for Java or Python. The third stack is for IoT Cloud platforms and is expected to provide horizontal scalability to support a large number of devices and vertically to support a variety of IoT scenarios and devices, including layers for device management, data management and storage, event management and analytics. A number of other middleware approaches that have been proposed for Cloud-Sensor integration are discussed in section 3.4.

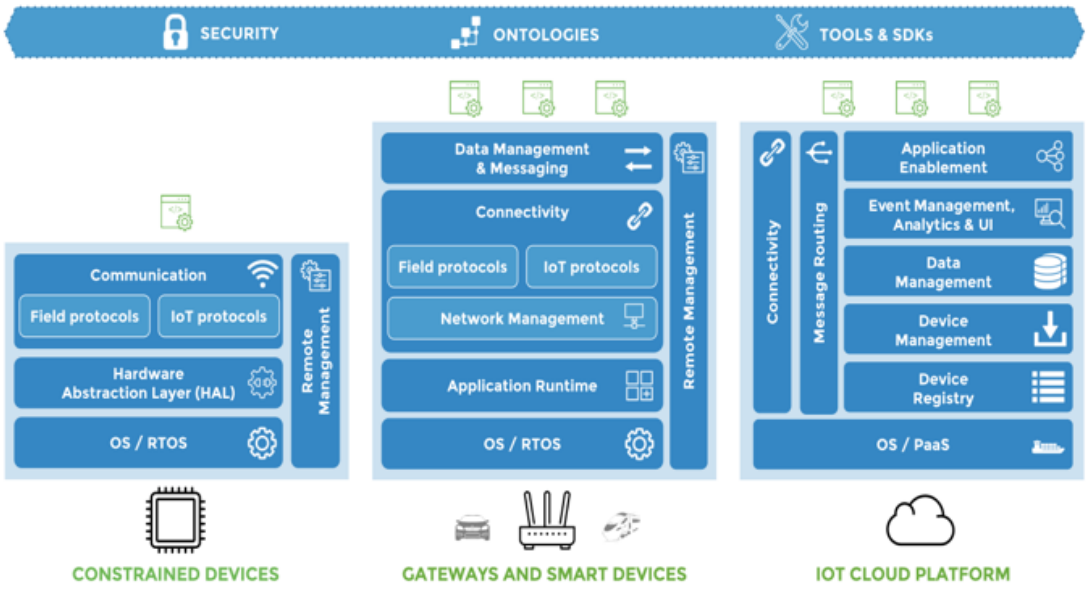

Figure 7 Eclipse IoT Stacks [69]

Sensation [70] is an example middleware solution for WSNs, where the sensor network as a whole is considered as an information source similar to a database. Its middleware acts as an integration layer between applications and networks using high level APIs. The API is supported by its Unified Sensor Language and a proxy in front of drivers for particular WSNs to hide device and network heterogeneity. While interesting in terms of allowing WSNs to collaborate, the approach of developing a proxy for each network and the requirement for a priori

configuration of network profiles (to conceal the underlying heterogeneity of WSNs) means that it does not meet a number of the requirements for a seamless, interoperable architecture. For example, while it is independent of particular node hardware, it does not provide a way to model a range of node functional capabilities. It is an extensible system, but does not provide abstractions in each WSN that could be used to handle the channel and environmental factors that will be encountered. It also does not provide a consistent means to exchange sensor information independent of the underlying technology, as this requires a proxy with a priori configuration rather than providing a means for the sensor node to advise other nodes and services of its capabilities.

Agent based middleware approaches have been proposed that use a set of services to provide abstractions and a language to compose sensing tasks from the services, where the service code can move across nodes autonomously. Such systems are suitable for monitoring moving and dynamic phenomena, but require particular node computational capability due to their complexity and the code mobility reduces node lifetime due to the additional network traffic [71].

TinyDB [72] and Cougar [73] essentially consider the WSN as a distributed database, with a table where each column represents a sensor reading or node data and a SQL like query language (extended for periodic or continuous requests) with nodes supporting aggregation of data. This is powerful, but is limited by its table based approach and relational queries, especially in handling events.

Directed diffusion is a method for data dissemination which can be considered as a superset of routing where the nature of the data and its usage is exploited as part of the routing of data. A WSN may be requested to provide information from a group of nodes or any node in a particular region, rather than a request being made to a particular node. Directed diffusion was designed to be more suitable in such cases [74] than ad-hoc routing techniques from IP networking, which generally use links between 2 unique addresses. Directed diffusion aims to extend the lifetime of the network by reducing message exchanges between nodes [75]. It does this by localizing exchanges within a limited network area, but still provides multipath delivery and adaptability to a minimal subset of network paths and also allows nodes to aggregate responses to queries.

Directed diffusion uses a publish and subscribe information model where a node expresses an interest in data items using a set of attribute-value pairs. This sink node broadcasts this periodically to its neighbours and the message propagates throughout the network. Each node keeps an interest cache with entries for each interest (containing timestamp, a "gradient" of data rate derived from interest requests and neighbour direction for reply, duration) and sets its sampling to meet the intervals in its interest cache entries. The interest cache is built based on interest requests from sink nodes which are used to create new entries or update the gradient, timestamp or duration of the existing entries. Nodes which can provide the relevant data will reply with it. The diffusion of data and establishing reply paths (as gradients) allows the discovery and establishment of paths between sinks interested in particular data and sources of that data. Nodes also keep a data cache, and they add to this when they receive data from a neighbouring node that matches an entry in their own interest cache (i.e. they received a request from a node interested in such data). An originating sink may have multiple paths to sources of data and it can favour (or reinforce) certain routes by sending its interest requests more frequently to one or more of its neighbours. Directed diffusion has the potential for energy savings, relatively high performance and stability for a range of network dynamics [17], but it is tightly coupled to a query on demand data model where applications can accept aggregated data and this limits its applicability.

3.2.3 Autonomic and Cognitive Architectures

Autonomic architectures are another approach to realizing complex, loosely-coupled, decentralised, dynamic systems as required for IoT. These architectures are characterized by self-configuration, self-healing, self-optimization and self-protection and aim to improve overall performance [76] by using cognitive approaches with information based on past experience. Cognitive entities in these architectures have reasoning capabilities and can cooperate, but may also act selfishly to exploit knowledge of their functionality.

Example frameworks for use in IoT, which are inspired by autonomic and cognitive principles include Cascadas, Focale, Inox and I-Core [76]. These frameworks are generally not concerned with creating an end-to-end holistic approach, but they are concerned with higher layer functions such as translating

vendor specific data into a vendor-neutral form or semantics around state transitions, reasoning engines, automatic management functions, perhaps using virtualisation. As such, they could be complementary to a holistic architecture, using it to gather data which it can then use for its higher-level functions.

3.3 Big Data and NoSQL Approaches

As the number of WSNs and nodes in IoT increases, the volume and variety of data to be collected, parsed and analysed increases correspondingly. The use of Big Data is well established in commercial and research environments to analyse large amounts of data in order to make timely decisions, e.g. in retail for analysing consumer behaviour and preferences. Big Data can be characterised by the 3 Vs of Volume (size of the data), Variety (range in type and source of data) and Velocity (frequency of data generation) [77]. There are a range of NoSQL data stores, such as simple key value ones like Redis [78] or document stores, e.g. MongoDB [79].

Apache HBase is an example of a Big Data store. It is a part of the Hadoop stack and uses the Hadoop Distributed File System (HDFS) to store its data. It is a distributed, versioned, column-oriented, data store, derived from Google BigTable [80]. HBase stores data into tables, rows and cells. Rows are sorted by row key and each cell in a table is specified by a row key, column key and a version, with the content held as an uninterpreted array of bytes. HBase can be considered suitable for WSN data based simply on its scalability and ability to store large amounts of replicated data. Its key value nature, handling of sparse rows and flexible data access provide other reasons for its suitability. The data access is provided by a rapid query using a get with a row key and a scan using an arbitrary combination of selected column family names, qualifier names, timestamp, and cell values. The support of sparse tables is appropriate for cases where not all WSN nodes can provide all the columns defined. Columns belong to a particular column family and are identified by a qualifier. Column families must be declared at schema definition time, but individual columns can be added to a family at run time. This provides flexibility to handle the varied data that sensors may send, i.e. whether a particular sensor sends all the attributes in a particular object. The associated MapReduce model has been shown to be appropriate for processing sensor data [81]. Modelling very large sensor network data as an ontology-based

Continuously Changing Data Object (CCDO) has been shown successfully for BigTable [82]. Section 6.5.2 illustrates how seamlessly the proposed holistic architecture can accommodate the use of Apache HBase to store sensor data.

3.4 Cloud and Sensor Platforms

The NIST has proposed three main Cloud service types/models of Infrastructure as a Service (IAAS), Platform as a Service (PAAS), and Software as a Service (SaaS) [83]. Their definitions capture the fact that these resources can be rapidly provisioned and released with minimum effort on demand, providing reduced upfront expenditure, but with high availability, fault-tolerance and what appears to consumers as infinite scalability.

Sensing as a Service has been proposed, with elements of an IAAS solution [8], but more often it is proposed as a kind of PAAS. The rationale for this interaction is to allow the huge storage, computing capabilities, data analytics, resource elasticity, and scalability provided by Cloud computing to form a key part of the IoT ecosystem. This allows more sources of data to be collected and for the data to be held for a longer time and to be processed by powerful Cloud based applications and Big Data techniques.

A number of Middleware approaches have been proposed for Cloud-Sensor integration. Sensor-Cloud [84] uses SensorML to describe sensor metadata and manages sensors via the cloud, rather than providing their data as a service. The OpenIoT [85] middleware platform comprises an IoTCloudController, a JMS style message broker, sensors (with a module to publish to OpenIoT) and clients (which subscribe to or consume sensor data). Another approach uses a data channel to hide the underlying network protocols and a Sensor Server on the wireless network's master node to filter sensor data and to deliver it to Cloud services [86]. This file based approach and use of existing technologies such as SSH tunnels, FTP servers presents a simple approach (with a connector-GUI and web server), but it is simple and limited in its flexibility. Another integration approach uses a content-based publish-subscribe model for event publications and subscriptions for asynchronous data exchange, requiring a gateway at the edge of the cloud to receive sensor data, a Pub/Sub broker to process and deliver events to registered users and a range of components to support SaaS applications [87], e.g.

System Manager, Provisioning Manager, Service Registry, Mediator, Policy Repository, Collaborator Agent. These middleware approaches to cloud integration all require specific application gateways/proxies at the edge of each wireless network and often their own sensor data definition.

Commercial offerings such as Amazon's IoT [88] and Google Cloud IoT [89] services allow users to upload their sensor data to the Cloud for storage, querying and analysis using their suite of Cloud databases, NoSQL stores and Machine Learning (ML) toolsets. These offerings use a proxy/gateway to provide the integration with a Cloud service, usually limiting it to a given Cloud provider and perhaps to a given environment on a device, e.g. Amazon's IoT service uses its own MQTT based software on a client device and uploads data to Amazon.

3.5 Fog and Edge Computing

The increasing amounts of data available from the Internet of Things (IoT) and the adoption of Cloud computing with its potential benefits in terms of scale, flexibility and cost have resulted in Cloud becoming a key part of IoT. There are, however, a number of issues with the approach of simply sending data to the Cloud and using it to handle all data from WSNs:

- Response Time – certain applications may require more rapid response time than the latency introduced by sending data to the Cloud will allow, e.g. connected vehicles.
- Intermittent Connectivity – this would prevent timely processing of the data (or cause it to be lost if device storage was exceeded) as required in application areas like telemedicine.
- Bandwidth – the amount of data sent by a large number of sensors connected to a given WAN link may exceed the bandwidth available.
- Device Connection – many devices in the Internet of Things (IoT) will need to be connected directly to each other, e.g. wearable health monitoring devices, connected vehicles.
- Data Security and Privacy – regulation may limit the countries or the data centres in which the data can reside, e.g. health data may require specific physical security guarantees.

One method to address these concerns is to push some of the applications to the edge of the network and process data there, possibly using concepts from mesh computing, grid computing or peer-to-peer computing.

Fog computing aims to tackle these challenges and “Fog Computing is a highly virtualized platform that provides compute, storage, and networking services between end devices and traditional Cloud computing data centres, typically, but not exclusively located at the edge of network” [90]. The processing at the edge may analyse, filter or aggregate the data and subsequently send it to the Cloud for further use. [91] gives the example of cameras in an autonomous vehicle which capture a huge amount of video data and which must be processed in real time to yield good driving decisions, making the response time from the Cloud too long. Furthermore, it also points out that a large number of autonomous vehicles in an area would further increase the pressure on network performance and reliability.

Such devices at the edge may both consume from and produce data to the Cloud, as well as load balancing that traffic. In such a federated system, a service may be executed using components running in different networks/providers, requiring fog and edge computing components to be interoperable at the level of providers and architecture models and interfaces.

Fog and edge computing essentially differ in the emphasis on how close the processing is to the source, where edge computing performs computing on an edge device like a programmable controller and fog computing performs it at the local network level where the processing can be done by a gateway or specialized node, although these distinctions are quite fluid.

An edge device is a computing or networking resource somewhere between the data sources and Cloud services, where the end device may both consume and produce data and provide storage/caching, processing on data sent to and from the Cloud, as well as load balancing that traffic. The range of entities considered in edge computing can be seen by the classes of edge computing presented in [92]. These classes are “resource rich servers deployed close to the end-devices”, “heterogeneous nodes at the edge” and “federation of resources at the edge and centralised data centers”.

As fog and edge computing are still emerging areas, it has a number of challenges to be addressed:

- Scalability - Individual edge systems will manage the data of a particular set of nodes which will have to scale as more nodes are added, but the overall system will also have to scale to manage, deploy and run large numbers of applications as more edge networks are added to cater for billions of IoT devices. Ideally, the edge networks will be able to use local resource pooling to achieve better scalability locally.
- Heterogeneity - An edge system should handle the storage, computational and operational requirements of heterogenous nodes and services, e.g. the different data formats used by devices.
- Management - Nodes and edge systems will require discovery and monitoring for the Cloud services to be aware of their status. A key question is whether this will be orchestrated in the Cloud and to what degree the nodes and edge systems will be autonomic and decentralised. Other management areas such as application provisioning and task scheduling, resource management and cloud/edge federation, including content storage/distribution have to be considered depending on the degree of centralisation used.
- Data Security - Edge nodes will have different capabilities and this will have to be considered in where data is to be stored or processed.

It is, however, important that the platforms, applications or services developed or configured for edge computing contribute to the seamless interoperability desired in IoT and not create more islands of isolated data and services. Such an expansive view of fog and edge computing has been proposed as Osmotic computing [93] to support Internet of Things (IoT) services and applications at the network edge. This paradigm is based “on the need for a holistic distributed system abstraction enabling the deployment of lightweight microservices on resource-constrained IoT platforms at the network edge, coupled with more complex microservices running on large-scale data centres”. It proposes to use the increase in resource capacity at the network edge to create edge micro data centres to form a federated environment of public/private cloud, edge cloud and devices with microservices in both, including a microservice engine to deploy containers running microservices on IoT and edge devices. It anticipates the use of an

interoperability layer for remote orchestration of heterogeneous edge devices, e.g. exploiting Software Defined Networking (SDN) and Network Function Virtualization (NFV) capabilities, accessible through an API.

Another proposed use of the increased processing power that can now be made available at the edge is to provide mobile edge computing for IoT to handle data streams at the mobile edge [94]. In this approach, each base station is connected to a fog node to provide local computing resources and a proxy Virtual Machine (VM), which collects, classifies, and analyses the raw data streams from devices, converts them into metadata, and transmits the metadata to the corresponding application VMs (owned by IoT service providers). A Software Defined Networking (SDN) based cellular core is used to forward packets among fog nodes.

The OpenFog Consortium (amalgamated with the Industrial Internet Consortium) [95] published an OpenFog Reference Architecture [96] as a basis on which to develop and test an open fog-enabled architecture. Such a model is also required if developers are to be able to handle the heterogeneity of fog computing and IoT. Figure 8 shows the OpenFog view of an N-tier IoT environment. It illustrates how the volume of data is reduced as the intelligence derived from the data is increased at each level. The OpenFog Reference Architecture has the following layers:

- *Devices* (sensors, actuators, cameras).
- *Monitoring and Control* (control logic using the sensor telemetry, e.g. to generate alerts and events).
- *Operational Support* (operational analytics).
- *Business Support* (such as large-scale historic analysis).

The three upper layers may be deployed only on fog nodes (e.g. where security concerns may rule out Cloud) or only on cloud nodes (e.g. where the physical infrastructure may not support fog nodes).

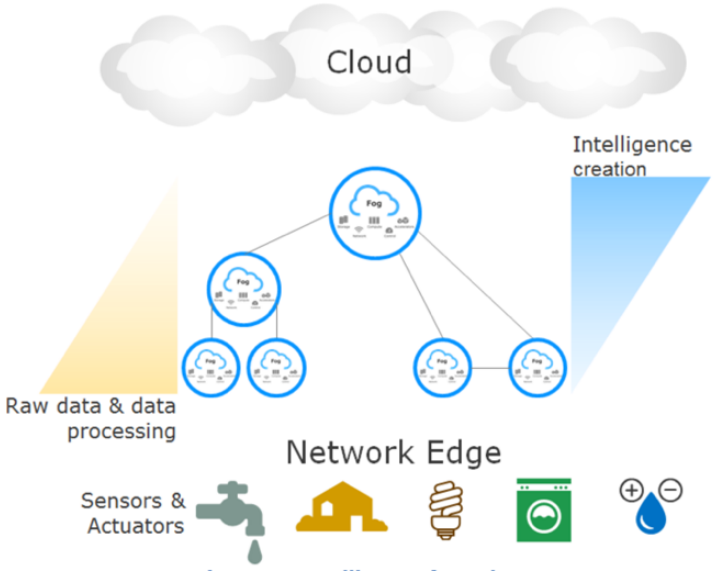

Figure 8 OpenFog Architecture Scenario [96]

Figure 9 shows a more detailed architectural description, including the cross-layer perspectives. It shows three views of the architecture - the “Software” view in the top three layers, the “System” view in the middle layers and the “Node” view in the lower layers.

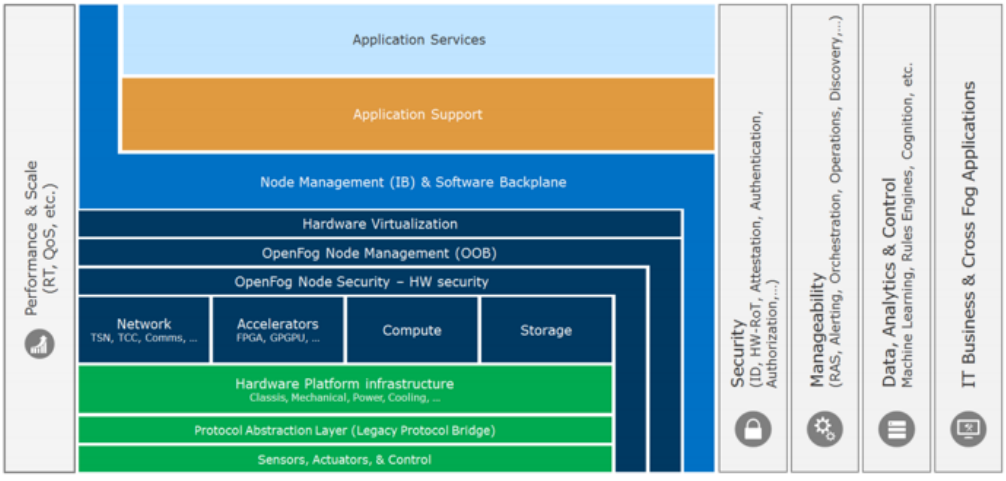

Figure 9 OpenFog Reference Architecture Description with Perspectives [96]

The OpenFog Reference architecture considers “Sensors, Actuators, and Control” as hardware or software-based devices, where several hundred or more of these items could be associated with a single fog node. Some of these nodes may have significant processing capability and are able to implement some basic fog

functions. The protocol abstraction layer exists to bring these devices under the supervision of a fog node so that their data can be provided to higher layers. The OpenFog Reference Architecture states that future versions will describe "Minimum Viable Interfaces", with more detail about the protocols and abstraction layers. It currently identifies protocols such as CoAP and MQTT for node-cloud and node-node communications.

It is worth mentioning that the emergence of 5G networks overlaps with Fog and edge computing in ways yet to be fully determined. For example, [97] considers that powerful nodes will be required at the edge of the network for 5G in order to offload traffic from the core, adjust the network resources for application data flow and to process the raw information from sensors/devices. Furthermore, it highlights the role of software based implementations and virtualisation (probably in the Cloud) to provide cost and deployment flexibility. [97] also considers the challenges for middleware from 5G are cloud-based Big Data management, interoperability of heterogenous devices and applications, scalability, context-based smart services provision, dynamic device discovery and management and security and privacy.

3.6 Tuple Approaches

A tuple space is a distributed computing concept, which provides a repository of tuples (a finite list of elements) available for concurrent access into which producers post their tuples and from which a consumer reads those tuples it wants. Linda [98] is an example which uses a tuple space model of parallel programming and considers its data objects as tuples and uses a small set of simple primitives:

- **in** atomically reads/removes a tuple from tuple space
- **rd** non-destructively reads a tuple from tuple space
- **out** writes a tuple into tuple space
- **eval** evaluates tuples, creating processes if required and writes the result into tuple space

The concepts behind Linda can be seen in the following: "If two processes need to communicate, they don't exchange messages or share a variable; instead, the data producing process generates a new data object (called a tuple) and sets it adrift in

a region called tuple space. The receiver process may now access the tuple. Creating new processes is handled in the same way: a process that needs to create a second, concurrently executing process generates a “live tuple” and sets it adrift in tuple space. The live tuple carries out some specified computation on its own, independent of the process that generated it, and then turns into an ordinary data object tuple [99].

Javaspaces [100] extended these ideas to Java objects. The decoupling in time and space of tuple space communication enables interactions where applications can be added/removed independently and do not have to be available simultaneously to transfer data between themselves. The tuple space approach allows different processes to use a limited number of simple operations to insert, read, and withdraw tuples from a tuple space and provide asynchronous notifications for data of interest being added to the shared tuple space.

TeenyLIME [101] is a high level tuple space based approach, derived from concepts used in Linda. TeenyLIME is built on top of TinyOS and inherits the nesC component model and can run on constrained devices. TeenyLIME also provides WSN-specific features, e.g. to hold system-level information about neighbours. A node’s local tuple space is only shared with the nodes within communication range. TeenyLIME has been deployed in a real-world application which has shown the usefulness of a tuple space approach in WSNs [102]. LighTS [103] is part of the LIME environment and provides a reduced tuple space holding context (location) information using the same primitives. LIME extended the local tuple space on every node by merging them into a federated tuple space into which tuples can be added/removed, when the nodes are in range of each other, i.e. tuples can only be exchanged when the sender of a tuple space operation is in range of the device offering the requested tuple. LIME is implemented in Java, limiting its applicability to more capable nodes.

3.7 Peer To Peer (P2P) Systems

P2P systems are used for file sharing, communication, collaboration, computation and distributed storage/databases. The most commonly known P2P systems are used primarily for music file sharing and sharing storage and bandwidth, such as BitTorrent systems. Applications have been driven by technology advances

allowing greater storage and bandwidth, enabling a home computer's processing and storage to be used for shared computing, such as SETI@Home. In a P2P network, these home computers connect to each other to form groups and collaborate to become clustered computers or shared filesystems, effectively a robust distributed system composed of inexpensive computers in unrelated administrative domains. Although the dominant mode of the Internet is client-server where browser clients access content servers, the original Internet was designed as a P2P system where it provided a communication means for (trusted) computer systems to share resources with each other as equals and indeed Usenet's decentralised model of control shares aspects of P2P systems [104].

One of the most appealing features of P2P for a large distributed system is the potential scalability that some P2P architectures provide, where the overall system capabilities (processing and storage) increase as the number of peers increase. This in contrast to some client/server architectures where increasing the number of clients may overload servers.

Napster was an early file-sharing P2P system, but it actually used a centralized server. Gnutella used a distributed file location and decentralised file lookup, but its use of multicasting a request from a node to all neighbours provided robustness at the cost of limited scalability (even with a TTL for a request), because of the bandwidth consumed by broadcast messages and the computing resources used by the many nodes handling these messages [104]. Another P2P system called Freenet [105] was designed to allow the exchange of files with the original source of the file remaining anonymous.

P2P architectures can be categorised based on the degree of centralisation, i.e. whether the architecture relies on one or more centralised servers as well as the edge nodes/peers. Three categories can be identified [106]:

- **Purely decentralised –** all nodes perform the same tasks, acting as servers and clients as appropriate. There is no central coordination of their activities, e.g. Freenet. The difficulties with these P2P systems include ensuring data consistency, manageability (as a completely ad-hoc self-managed system it may not align with wider policies such as charging, security), overhead as a result of the interaction between nodes to co-

ordinate (and exchange information) and reliability (may provide only a best-effort service).

- **Partially centralised** - this is similar to purely decentralised systems, except that there is a hierarchy where some nodes (supernodes) perform a larger role, such as holding central indices for files shared by local peers. These supernodes are not single points of failure because they are dynamically assigned and the role can be taken by other nodes in case of failure or attack. These supernodes will take a higher load and as such are likely to be more expensive and more capable compute nodes, e.g. later versions of Gnutella.
- **Hybrid decentralised** - these use a central server to maintain directories of the shared files stored on the edge nodes. The end-to-end interaction is between two peers, but this is set up using these central servers to look up the respective nodes holding the files. These central servers do, however, present a single point of failure and as such are vulnerable to failures, attacks or censorship. It is arguable whether these are “real” P2P systems, given this usage of a standard client-server relationship, e.g. BitTorrent when not using a DHT tracker or Napster.

P2P can also be categorised by the extent of structure in the topology in the P2P network. The topology of this overlay P2P networks can be very different to the physical network connecting the different nodes. P2P can be classified as follows using network structure:

- **Unstructured Networks** - the location of data is independent of the P2P topology. This lack of structure means that such systems can handle nodes entering/leaving relatively easily, but searches for data consist of nodes being probed (in a random manner) for the data being requested and this limits scalability as many nodes may need to be queried.
- **Structured Networks** - this type of network addresses the scalability issues of unstructured systems by controlling the overlay P2P network's topology by placing data (or pointers to it) at specific locations based on some criteria to form a distributed routing table. This should result in the efficient routing of queries to the node with the data. Such networks are suitable for queries where a complete identifier of the data object is given

in the request. They do, however, incur overhead to maintain their structure, particularly where nodes enter/leave at a high rate.

- **Loosely structured** - data is placed using routing hints, rather than exact specifiers. This reduces the maintenance overhead, but also means that not all searches will succeed. Freenet is an example of such a network.

In general, structured P2P overlay networks, e.g. implemented with a Distributed Hash Table (DHT) provide efficient data storage and lookup, whereas unstructured overlays rely on flooding or multicast approaches for message routing.

3.7.1 Freenet

Freenet is a purely decentralised and loosely structured system, operating as a self-organising P2P network. Freenet was designed to hide the origin or destination of a file passing through it, with the responsibility for data on a node being separated from the operator of that node. This was done to support the exchange of information in countries where this may be difficult or dangerous for those originating the information [105]. Providing this anonymity means that Freenet does not associate a file with any predictable server or have a predictable topology of servers. This also means that unpopular (infrequently accessed) documents may disappear from the system as there is no server responsible for maintaining replicas and so a search may traverse a large fraction of the Freenet network [104]. Also, as it did not check the files being shared, Freenet was vulnerable to viruses and infiltration attacks.

Unlike BitTorrent, Freenet is a file-storage and not a file-sharing service and files are pushed to other nodes for storage not only when these nodes request them. A Freenet node has a dynamic routing table containing the addresses of other nodes and the file identifier keys and it also has its own local datastore, which is available to other nodes to read and write.

Freenet Messages have an identifier (for loop detection), a hops-to-live, source and destination, and a type, which is **Data request** (with a Key field), **Data reply (**with a Data field), **Data failed (**with fields for Location and reason) or **Data insert (**with fields for key and data). A user searches for a file by sending a request message with the file identifier key and a hops-to-live.

A new node joins the Freenet network by discovering the address of one or more existing nodes and sending messages. To add a file (and so announce its presence), this new node sends a **Data Insert** message to add a file to its own node and this message holds the binary key calculated for the file and the hops-to-live. When received, this key in the insert message is checked and if in use, the node returns the associated file, but if not found the node looks up the nearest key in its routing table and forwards the message to the corresponding node.

Freenet does not broadcast requests and requests for keys are passed from node to node with each node deciding where to send the request to. If a node has stored a requested file, then it sends the data back to the requester, otherwise it forwards the data to the node it knows about with the "closest" file identifier key. The hops-to-live in the message avoids long forwarding chains.

A Freenet node stores a file, the next hop where it forwarded the file and the file identifier key of the requests passed on by it. This means that on receiving a request failed message from the node it forwarded to, it can forward the request to the "next best" node from its routing stack. When all nodes in its routing table have replied with request failed, this node itself will send back a request failed message to the node it got the request from. On the other hand, if the file is found at a node, the reply is sent back by the path which forwarded the request. Importantly, all of these intermediate nodes will cache the actual data in the data reply message, meaning that a reply to a future request will use the cached data.

3.7.2 JXTA

JXTA [107] is a P2P system with a suite of protocols specifically designed for ad hoc, pervasive, and multi-hop P2P computing. It allows peers to form self-organized and self-configured peer groups without a centralized management infrastructure. Its use of Java, its heavy use of strings and XML limit its applicability in resource constrained devices, but the following abstractions are useful to consider in the context of an end to end system including WSNs:

- Peers - JXTA considers any network device as an autonomous peer having a unique identifier and interacting with a small number of other peers. Peers may join or leave the network at any time. It has the concept of a peer group, with a unique identifier, for peers with common interests.

- Network Services - peers cooperate to publish, discover and invoke network services, either peer services accessible on that peer only or peer group services running on multiple peers in a group. Peers discover network services via the Peer Discovery protocol.
- Pipes - these are a network abstraction (a similar concept to a Unix pipe) over the peer endpoint transport and connect one or more endpoints. A pipe can operate in a point to point manner or in a propagate manner (connects one output pipe and multiple input pipes).
- Messages - information is packaged as self-describing messages defined in XML, using an envelope to transfer data with an arbitrary number of named sub-sections holding any form of data.
- Advertisements – these describe network resources, such as peers, peer groups, pipes and services. Advertisements are published with a lifetime that specifies their availability and can be republished to extend it.

The JXTA protocols do not require a particular network transport or topology or the use of a particular authentication, security or encryption model, as the format of JXTA messages enables the carrying of metadata, such as credentials (an opaque token to be presented each time a message is sent), digests, certificates and public keys. Messages may also be encrypted and signed for confidentiality and refutability. The JXTA protocols standardize the manner in which peers:

- Self-organize (publish, discover, join, and monitor) into peer groups. Peers wanting to join a peer group need to discover at least one member of the group and request to join. This request is either rejected or accepted by the collective set of current members or a membership service may enforce a vote of peers or elect a member to accept/reject new membership requests.
- Advertise their own and discover other peer's network services and resources (CPU, storage, databases, documents etc.).
- Communicate with each other and route messages across multiple network hops to any destination in the network (each message carries with it an ordered list of gateway peers through which the message might be routed).

3.7.3 Distributed Hash Table Based Networks

Distributed Hash Tables (DHT) are used in a number of structured P2P systems. They use consistent hashing algorithms, which ensure that the routing of requests

is deterministic with an upper bound on the number of hops, although this implies some overhead to manage the peers in the overlay. They provide efficient routing without centralized control, e.g. BitTorrent's [12] use of Kademlia [13].

A hash-table is suitable for distributed lookup as it only requires that data can be identified using unique numeric keys, and that nodes store keys for each other. Nodes store information about neighbouring nodes, forming an overlay network and route messages in the overlay to store and retrieve keys. Data items are inserted into a DHT and found by specifying a unique key for a data item. A DHT algorithm determines which node is responsible for storing the data associated with a key, using a lookup(key) call that returns the node identity (perhaps its IP address). In the case of storing files, the key may be derived by applying a hash function to the file name and inserted into the store using that key. A subsequent lookup(key) builds the key using the same hash applied to that file name. DHT lookup algorithms have to address the following [14]:

- Mapping keys to nodes - nodes and keys are mapped using a hash function into a string of digits. The node for a data item's key is assigned based on its identifier being the "closest" (perhaps numerically or with the longest matching prefix) to the key.
- Forwarding a lookup for a key - on receiving a lookup for a key, a node must be able to forward to a node with a "closer" identifier to that key (not necessarily the final destination, but closer to it).
- Building routing tables – different forwarding rules and different information is held on successor nodes depending on the DHT. Each node keeps a routing table for selected nodes, where it holds their identifier and so can determine which one to forward to, based on their closeness to the key (or if it is the closest node itself).

The following sections consider a number of DHTs and their scalability to large numbers of nodes, low latency to find keys, ease of maintaining node routing tables and balancing the distribution of keys. They differ in how they build and maintain their routing tables as nodes join and leave. They may provide bounded times for data lookups by limits on the number of hops meaning that response times could be guaranteed in wired sensor networks (although the higher probability of link failure limits this in wireless networks). Ensuring that the number of hops in a P2P overlay network maps to bounded latency does require structuring the P2P overlay based on the actual routing topology [108].

In terms of the dynamic nature of the network and the frequency of join/leave events, it can be seen that the effect of relatively frequent node joins and departures in large systems could end up dominating overall performance and the following sections describe how the DHTs handle the join/leave events

3.7.3.1 CAN

CAN [38] partitions a fixed d-dimensional Cartesian coordinate space into hyper-rectangles, called zones with a node being responsible for a zone. A key is mapped onto a point within a zone and its data is stored at the node for that zone. A node's routing table holds all of its neighbours in that coordinate space, i.e. they share a (d-1) dimensional hyperplane. A node forwards messages to its neighbour closest in the coordinate space to the target node storing the key. With d dimension CAN knows of d nodes and has $O(dN^{1/d})$ lookup and $O(dN^{1/d} + d \log N)$ join.

When a node j wants to join the network, it chooses a random point in the coordinate space and asks an existing node to find the node r with the zone containing that point. That node r then checks the size of its neighbour's zones - the neighbour with the largest zone (or itself if it has the largest) splits its zone in two, assigning one of the halves to the joining node j. The new node j initialises its routing table to contain all of node r's neighbours (except itself) and announces itself to its neighbours which update their routing tables to include it. The reverse occurs when a node leaves the network voluntarily as it hands its zone to one of its neighbours. If a node leaves involuntarily, the neighbour with the smallest zone takes over its zone. Multiple failures can cause fragmentation resulting in some nodes handling a number of disjoint zones and CAN uses a background algorithm to combine adjoining zones to the same node.

3.7.3.2 Chord

Chord [108] uses a one-dimensional space to assign identifiers randomly for both keys and nodes and this space wraps to form a circle. The node responsible for key k has the identifier most closely following k. Chord was designed for use over the Internet and its focus is on robustness and correctness. Each node holds a “finger table” containing the IP addresses of nodes halfway, quarter way and

successive powers of 2 around the ring from it, so the finger table holds log N entries for N nodes.

A node forwards a request for key k to the node in its “finger table” whose identifier is the highest one less than the key k and the selection may also consider the link latency. The power of 2 structure of the finger table means that the query is forwarded to at least half of the remaining identifier space to the node holding k and so there will be O(log N) messages sent by Chord.

An example lookup for key 81using Chord is shown in Figure 10, using reduced node identifiers. It shows the use of the finger table, until node 71 determines that its direct successor node 81 is responsible for that key 81 (and an associated value) and then node 71 forwards the request to node 81.

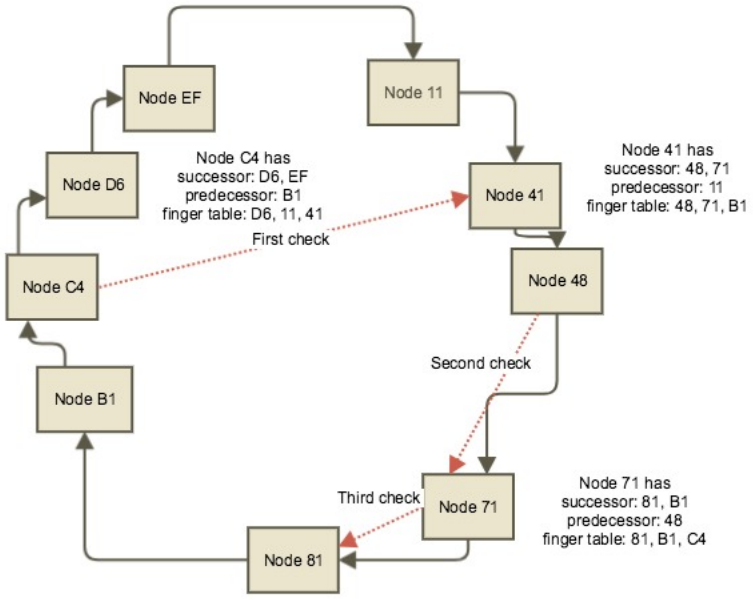

Figure 10 Example of Lookup in Chord

When a new node j wants to join, it requests an existing node to look up the identifier for node j and then its own and its predecessor update their successor lists. The updates to the finger tables for node j and existing nodes are done in the background rather than at joining (as they are done to improve performance and not for correctness of routing). The new node j must also acquire the existing data

associated with the keys it is now responsible for and it does this by fetching it from its successor (which previously held them).

Chord handles node failures as nodes also store the IP address of a number of nodes and so packets can be forwarded to one of these successors even if the node selected using the finger table is not available. Hence forwarding failure would only occur if all the successors and all those in the finger table failed simultaneously. For this reason, node identifiers are assigned randomly and nodes in the successor list are unlikely to suffer independent, simultaneous failures.

3.7.3.3 Pastry

Pastry [109] assigns a node a random identifier indicating its position on a circle of identifiers. Messages are routed to the node identifier (numerically) closest to the search key. Each node keeps a routing table with each row holding identifiers for nodes sharing a number of digits in their identifier with this node. Nodes also keep a “leaf set” (analogous to Chord's successor list) of nodes either side of its identifier. The leaf set is checked first and if no match, then it checks the routing table for a node identifier with a longer shared prefix. If such a node is not in the routing table, then it tries a node with a shared prefix at least as long as this node’s and which is (numerically) closer to the key – this should only occur at the desired node or its immediate neighbour (otherwise such an entry would have been in the leaf set). Each forwarding of the message increases the number of shared digits between the key and the node identifier. This use of different routing algorithms is problematic according to [13] as nodes close by the second can be quite far by the first. Pastry also uses heuristics based on network proximity, e.g. number of hops, to forward a query when there is more than 1 possible node in the routing table.

A new node is given a random identifier and builds its leaf set and routing table using information from the node with the identifier closest to its one. When a node leaves, only the leaf sets of effected nodes are updated and the routing table information is only updated when a node tries to reach the departed node.

3.7.3.4 Tapestry

Tapestry [110] maps node and key identifiers into strings of numbers, but has a greater focus on proximity in a network sense than CAN, Chord or Pastry with the goal of reducing latency in forwarding queries. This comes at the cost of increased

complexity, particularly when handling nodes joining and leaving. Tapestry forwards queries to nodes closer to the target a single digit at a time. This requires nodes to maintain lists of nodes matching its own prefix but differing by the next digit and to do this for each prefix of its own identifier. A node 697512 would keep lists of nodes with prefixes 6x (where x is not 9) to forward to a node matching a 2nd digit of a query and lists up to 69751x (where x is not 2). Maintenance of these lists by all nodes in a dynamic network can be seen to be demanding. For example, a query for node 697512 could be forwarded from 697892 to a "closer" node such as 697598 with 4 matching initial digits.

3.7.3.5 Kademlia

Kademlia is a peer-to-peer system to store and lookup key value pairs [13]. Kademlia keys are opaque, 160-bit entities and each peer has such an entity as its identifier. For two such 160-bit identifiers, Kademlia defines the distance between them as their bitwise exclusive or (XOR), i.e. closer nodes have more common bits in their prefix. Where $d(x, y)$ is the distance between two points, $d(x, x) = 0$, so that $d(x, y) > 0$ if $x != y$. The use of XOR also means that $d(x, y) + d(y, z) >= d(x, z)$. Kademlia is also symmetric in that $d(x, y) = d(y, x)$ for all x and y, whereas Chord is not symmetric and Pastry is. Another important property of Kademlia is that it is unidirectional, i.e. for point x and for any distance $D > 0$, there is exactly one point y such that $d(x, y) = D$ [13]. This means that all lookups for the same key will converge along the same path even if the lookups came from different nodes. This means that caching along the path will reduce the issue of hot spots, as the cached value can be returned for subsequent requests.

In a fully populated binary tree of 160 bit identifiers, the distance between two identifiers is the height of the smallest subtree which contains both identifiers. If the tree is partially populated, the leaf closest to a given identifier x is the leaf with an identifier sharing the longest common prefix with that given identifier x. In the case of partially populated trees there may be empty branches giving more than one closest leaf. In that case Kademlia, flips the bits in x corresponding to those empty branches to give x' and selects the leaf closest to x'.

A node using Kademlia can find its closest peer nodes and route queries, while every message exchanged has information which can be used to update the address details for nodes. Its key benefits are:

- it minimises the number of configuration messages required for nodes to discover each other, as this information is also carried in messages used to lookup keys, e.g. every message includes the sending node's identifier. XOR is symmetric, so Kademlia peers will receive lookup queries from the same distribution of nodes that are in their routing tables and this also provides benefits if used with caching.
- nodes can use metrics to route queries through low-latency paths, based on storing ⟨key, value⟩ pairs on nodes with identifiers "close" to the key.
- it uses the XOR based routing algorithm from start to finish in locating nodes close to a particular identifier.
- it avoids timeout delays from failed nodes by using parallel, asynchronous queries (not necessarily useful in a WSN).

Figure 11 shows an example binary tree for Node 0011, with the subtrees of interest to nodes with identifiers 0011aaaabbbbccccdddd, i.e. those subtrees derived from changing a bit in the 0011 prefix.

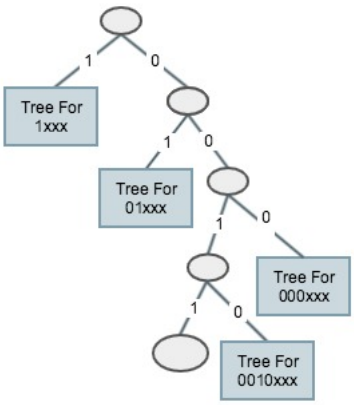

Figure 11 Kademlia Binary Tree Example

Kademlia nodes keep a list of ⟨IP address, UDP port, Node ID⟩ triples for nodes of distance between 2^i and 2^{i+1} from itself for each $0 \leq i < 160$. These lists are termed k-buckets in Kademlia and they are sorted by time last seen. Lists are allowed to

grow up to a system defined limit of size k, with least recently seen entries replaced with the proviso that active nodes are not replaced by newer ones. This was based on behaviour seen in Gnutella, where it was noticed that the longer a node has been up, the more likely it is to remain up another hour. The value of k is chosen for a particular use case so that k nodes are very unlikely to fail within one hour (the refresh period) of each other, i.e. the value of k is set according to the number of simultaneous failures anticipated in the refresh period, because after a refresh, a node has k closest nodes to it or every node within range if there are less than k nodes within range [13].

Kademlia uses four messages:

- PING – this checks to see if a node identifier is available
- FIND NODE – contains a target identifier which a node responds to with the ⟨IP address, UDP port, Node ID⟩ for the k nodes closest to that target identifier or all its nodes if it has less than k in its buckets.
- FIND VALUE – contains a target key identifier to which a node responds with a value from an earlier STORE or the ⟨IP address, UDP port, Node ID⟩ triples for nodes as with FIND_NODE
- STORE – this sends a key value pair to a node for storage

On receiving a message, a node updates the times in the k-bucket for the sender's node identifier and the contents of the bucket. A node identifier already in the bucket is moved to the list's tail. A node identifier not in the bucket is inserted if the bucket is not full, i.e. holds less than k items. If the bucket is full, the least recently seen node is sent a PING message and if it responds, then it is moved to the tail of the list (as for any received message) and if not it is replaced with the sender of the message that started this process placed at the tail. It is expected that normal message usage between nodes will keep the buckets up to date, but a node will also refresh a bucket in which no identifier has been looked up in the last hour, by sending a FIND_NODE for a random identifier in the bucket's range.

A node wanting to join the network does a lookup for its own node identifier to a known node in the network, resulting in its identifier being inserted into that known node's bucket, and receiving address triples for other nodes in the

response. It continues informing other nodes about itself and adding entries from responses into its own buckets until no closer nodes can be found.

Kademlia “node lookup” is the term for finding the k closest nodes to an identifier [13]. This begins with the initiating node picking α nodes (from its closest non-empty k-bucket or if that bucket has too few it uses the α closest nodes in its buckets) and sending a FIND_NODE to each of them (can be done in parallel), where α=3 is suggested. It then recursively sends FIND_NODE messages to nodes it has learned about from previous messages, picking nodes it has not sent requests to. If this does not produce a node any closer than the closest already seen, the initiator resends the FIND NODE to all of the k closest nodes it has not already queried and this lookup finishes when it has received responses from the k closest nodes it has seen (possibly from more than one bucket). As per [13] using α = 1 this lookup algorithm resembles Chord’s in terms of message cost and the latency of detecting failed nodes.

STORE operates in a similar manner, with STORE messages sent to the k closest nodes to the key. A node re-sends the STORE to refresh those values and a frequency of every hour is suggested, with a limit of 24 hours before a value is removed.

FIND_VALUE is sent similarly to the k closest nodes, but it stops when any node returns the value. On a successful lookup, the requesting node also stores the ⟨key,value⟩ pair at the closest node it observed to the key that did not return the value. Furthermore, the unidirectionality of the topology means that future searches for the same key are likely to hit cached entries before querying the closest node. It sets an expiry time in the cache exponentially inversely proportional to the number of nodes between the current node and the node whose identifier is closest to the key identifier, rather than using LRU eviction as it does not know how many values it should store.

Kademlia must republish key-value pairs periodically for nodes which may join or leave (in the case of mobile or failing nodes) the network. Kademlia republishes each key-value pair once an hour. It does not do this by each of k nodes performing a node lookup followed by k-1 STORE messages every hour as this would be expensive, particularly as the number of nodes increases. It optimises

republishing by each node assuming that a STORE was also issued to the other k-1 closest nodes and so this receiving node will not republish the key-value pair in the next hour. It is expected that the republish periods are not synchronised, so that only one node will republish a given key-value pair an hour. Another optimisation is for a node to refresh all k buckets, before republishing key-value pairs as it will be able to determine the k closest nodes to that key that are still in the network. When a new node joins the network, other nodes should issue a STORE to send relevant key-value pairs into the network, but this can be optimised to avoid redundant STORE messages by a node only sending a key-value pair if its own identifier is closer to that key than the identifier of other nodes that it knows of.

3.7.4 BitTorrent

BitTorrent [12] is a protocol for efficiently distributing static data, primarily files, broken into pieces with a SHA-1 hash. It uses a URL to identify content and is designed to integrate seamlessly with the web. A metadata file (.torrent) is distributed to all peers with a tracker reference, the SHA-1 hashes of all pieces and a mapping of the pieces to files. A swarm is the set of peers taking part in distributing the same files. The tracker is a central server, which holds a list of all peers in the swarm, where a peer joins a swarm by asking the tracker for a peer list and then it connects to those peers. The use of a central server as a tracker, however, is a single point of failure and may create a bottleneck for publishers. For this reason, Trackerless Torrents were added and one of the approaches for this is a DHT based on Kademlia over UDP. BitTorrent refers to peers as being a client/server listening on a TCP port that implements the BitTorrent protocol, whereas it terms a "node" as a client/server listening on a UDP port implementing the DHT protocol.

In the Trackerless torrent, BitTorrent peers include a DHT node, which holds the location of peers to download from using the BitTorrent protocol. The key is the info-hash (the hash of the metadata), which uniquely identifies a torrent and the value is a peer list of the peers in the swarm, containing the contact information for those peers [111].

BitTorrent wants only “good” nodes in the routing tables, where that means it has responded to a query from a node within the last 15 minutes or if it has ever responded to one of its queries and has sent it a query within the last 15 minutes. It is deemed questionable after 15 minutes of inactivity and bad after failing to respond to multiple queries. It uses buckets as per Kademlia, with the 15 minutes being used to determine the last seen recency, i.e. a node is selected to be pinged and possibly replaced if it has not sent or received any messages in the last 15 minutes. Similarly, buckets that have not been changed in 15 minutes should be "refreshed."

Peers supporting the DHT trackerless torrent set the last bit of the 8-byte reserved flags in the BitTorrent protocol handshake. On receiving a handshake indicating the remote peer supports the DHT, a BitTorrent peer sends a PORT message. On receipt of this, that peer should ping the DHT node on the received port and IP address of the remote peer and then handle the response as per Kademlia.

BitTorrent uses the KRPC protocol consisting of binary encoded dictionaries sent over UDP and there is no retry. There are query, response, and error messages. For the DHT protocol, there are four queries: PING, FIND_NODE, GET_PEERS and ANNOUNCE_PEER:

- A PING query has a 20 byte "id" string containing the sender’s node identifier and the response has an "id" containing the node identifier of the responding node.
- FIND_NODE has an "id" containing the node identifier of the querying node, and "target" containing the identifier of the node sought by the query. The receiver should respond with a key value pair of key "nodes" and value as a string containing the compacted node information for the target node or the k (8) closest good nodes in its own routing table.
- GET_PEERS retrieves the peers associated with a torrent info-hash. It has "id" for the node identifier of the querying node and "info_hash" for the info-hash of the torrent.
    - If the queried node has peers for the info-hash, it returns a key value pair of key "values" and a value consisting of a list of strings. Each string holds compacted peer information for a single peer.

    - If the queried node has no peers for the info-hash, it returns a key value pair of key "nodes" and “value” of the compacted node information for the k nodes closest to the info-hash in the queried node’s routing table.
    - a "token" key is always included in the response for use in a future ANNOUNCE_PEER query.
- ANNOUNCE_PEER - announces that the peer, controlling the querying node, is downloading a torrent on a port. It has four arguments: "id" containing the node identifier of the querying node, "info_hash" containing the info-hash of the torrent, "port" containing the port and the "token" received in a GET_PEERS response. The queried node receiving this ANNOUNCE_PEER must verify that the token was previously sent to the same IP address as the sender of this ANNOUNCE_PEER query and should store the IP address of the querying node and the supplied port number under the info-hash in its store of peer contact information.

Each BitTorrent peer announces itself with the distributed tracker by looking up the 8 DHT nodes closest to the info-hash of the torrent and sends an announce message to them. Those 8 nodes add the announcing peer to the peer list stored at that info-hash as above.

It is important to note that a BitTorrent peer uses the DHT to know more about “good” peers close to it, i.e. it favours the storage of peers close to it and does not store all peers. Peers that are deemed “good” (have sent a message to this peer within a specified period as above) are kept in the bucket on the assumption that good peers are long-lived and should not be replaced so long as they remain “good”. It builds up its knowledge of close peers by starting with a single bucket and splits that bucket when it is full of “good” peers and it is determined that a new peer is to be added. That determination is based on whether the new peer identifier is in range of the identifier for the peer holding the bucket and the first identifier in the next bucket, except in the case of it only having one bucket where a split is done into buckets for identifiers 0 to 2^{159} and 2^{159} to 2^{160}. Otherwise the split is done with a new bucket having a middle identifier of the existing bucket as its first identifier and peers allocated to each bucket according to whether their identifier is less than or greater than that middle identifier. If the determination above deemed that the peer is out of range, then the current bucket will not be

split and an existing entry will not be replaced. It is also important to note the lazy nature of entry replacement, i.e. if a bucket is full of bad peers, the bad entries are only replaced when a new add is received for an identifier within range.

BitTorrent uses PING to find nodes close to a peer when:

- adding to a bucket in order to determine what to do if the bucket is full, i.e. it pings a peer in the bucket that has not sent a message to this peer in the specified period and removes it from the bucket if it does not reply.
- starting up, when a peer sends ping
- refreshing a randomly selected peer identifier

It is worth noting that BitTorrent extended the messages in Kademlia and introduced its own separation of node identifier and info-hash, but that it retained the essentials of the node lookup and associated buckets. It also changed the refresh behaviour and set k to 8 as this was considered sufficient to reduce the probability of that number of nodes disappearing from the network within the refresh periods.

3.8 P2P in Wireless Sensor Networks

Section 3.7 showed P2P systems that can act in an autonomic, self-organising and dynamic manner (with no centralised authority). The following sections consider some P2P approaches that have been used in WSNs.

3.8.1 WSN as a Peer via a Gateway to Mobile Network

One approach is to use P2P as a means of interacting between sensor networks and gateways to a wider network. In an example of this approach [112], the sensor network is viewed as one peer in the P2P network and is represented by its gateway in the mobile network, allowing users to access each network over the mobile network. This is a similar topology to the network routing based RPL approach, but it also provides a Sensor Network Abstraction layer and defines sensors using a sensor profile consisting of an attribute profile for basic sensor features such as type and a data profile for the data formats. It provides P2P protocols for the sensor networks to publish available services, query all peer

sensor nodes and to search for available services using a Service Discovery protocol. The Sensor Network Abstraction layer provides a higher layer application with access to all sensors in a given WSN and an object based API to communicate with and control a sensor or group of sensors.

This approach is interesting as it confirms that sensor characteristics can and should be catered for using abstractions and supporting layers, but the assumed use of a sink to interface to an external network limits the scalability and deployment of the overall network (unless all the gateways shared networking and sensor abstractions).

3.8.2 Distributed Hash Tables in WSNs

The lookup times achievable by DHTs suggest that their use may be appropriate in sensor networks. Use of DHT based approaches in WSNs has been argued against for reasons of difficulties in topology mapping (the nodes that DHT connects logically as neighbours may in fact be physically far apart), the overhead of route maintenance (among all pairs of nodes), sensor names and their identifiers and the computational cost of computing DHTs. These arguments can be countered as follows [113]:

- the P2P overlay topology can be aligned with the physical topology (using virtual rings) so that the P2P neighbour is the physically closest node.
- Virtual Ring Routing (VRR) allows a DHT-inspired routing protocol to sit directly on top of the link layer, where nodes are organized into a virtual ring and every node maintains a small number of routing paths to its neighbours in the ring [114].
- it is usually a requirement in real deployments to assign a globally unique identifier to nodes (as also required by DHT) for reasons of network management/configuration and debugging (and expediency for data collection, event handling), rather than their being named using data/application attributes. It is important, however, to minimise the address size of network-wide unique sensor network addresses (possibly by assigning addresses dynamically from small, locally-unique addresses).
- DHT computation is within the capabilities of simple WSN nodes.

DHTs are location independent, unlike Geographic Hash Tables (GHTs) and so are more suited to applications where a priori knowledge of geographic boundary areas is not available or where the areas themselves change.

3.8.3 Tiered Chord (TChord)

Tiered Chord (TChord) [113] is a Distributed Hash Table (DHT) based P2P overlay for sensor networks with the goal of eliminating sinks and proxies. Chord was chosen for its predictable lookup time, its relative simplicity, its robustness as illustrated earlier and its efficient handling of concurrent node joins and failures (even though it is structured, which is often seen as an impediment in handling high levels of dynamicity in networks).

TChord used a SHA-1 hash on the unique 64 bit MAC address of each sensor node for its node identifier. It also generated metadata keys from the data stored on the nodes and hashed them to create key identifiers. TChord simplified the mapping of Chord onto Sensor networks by categorising nodes as master and slave nodes. The master nodes are connected in a ring with all messages routed clockwise. Every ring has a master node to hold information on its slave nodes and other O(logN) master nodes (in its finger table like Chord). Master node information for the other master nodes in the ring is updated on nodes leaving or failing and a master node's data is replicated on neighbouring masters.

A received query is handled by use of the finger table as with Chord, but using the finger table on the master nodes rather than the power of 2 approach used in Chord. If a master node or its own slaves cannot resolve the query, it uses its finger table to locate the master node with that key and forwards the query to the master node storing the data or to the master node closest to the target according to the finger table. In order to improve simplicity, slave nodes are not connected in a ring and do not hold information on their neighbours. Consequently, a slave node will check its data on receiving a query and if this does not match, it forwards the query to its master node. TChord was designed to co-exist with the SensorNet Protocol (SP), which sits between the network and data link layers to enable data-processing at each hop and not just at end points [115].

3.8.4 Service and Resource Discovery using P2P

Scalable service and resource discovery is a key part of being able to deploy applications and services in IoT. In CoAP, service discovery is the process used by a client to discover the end-points available on a server. A self-configuring P2P based architecture is proposed in [116] to automate this discovery with the aim of removing the need for human intervention. This architecture distinguishes between local and global service discovery. Local service discovery works within a single network and global service discovery works across different, perhaps geographically separate, networks. It was experimentally tested on Contiki for local service discovery and used a Java implementation for global service discovery. This paper also considers a number of previous service discovery approaches using RESTFul approaches or grouping nodes with border routers and a central border router, but they consider P2P to be the most scalable and robust.

This architecture uses a special “IoT Gateway” node at the boundary between a local WSN and a larger P2P overlay network. This gateway gathers information on the resources in that WSN using CoAP. The performance of global service discovery is dependent only on the size of the P2P overlay and not of the entire IoT containing all the nodes. The authors of [116] state that the approach of every peer being in the overlay is not suitable for IoT, as many nodes are constrained in terms of their processing capabilities. The IoT Gateway implements IP gateway and CoAP to HTTP proxy functionality, but also provides caching to reduce load on WSN devices and protects the constrained nodes in the WSN from denial-of-service attacks.

The Distributed Location Service (DLS) used in its global service discovery is based on a DHT, which provides a name resolution service for the binding between a URI for a resource and the information on how to access that resource. Interestingly this information includes an expiration time. In addition, they provide a Distributed Geographic Table (DGT) to locate information near a geographic location. This uses a structured overlay where geographically close nodes are neighbours in the overlay network. Their implementation of the local service discovery on Contiki in linear and grid topologies showed that the query time on the server was almost constant, but that the time for a client to receive a response depended heavily on the number of hops involved. The global service

discovery was a logarithmically increasing function with the number of peers, indicating the scalability of their approach.

A similar view regarding the limitations of constrained devices is expressed in [117], where the WSN and external P2P network are treated separately due to the “ad-hoc, costly and difficult-to maintain, scale and extend” nature of WSN applications. A programming abstraction based on “feedback loops” to describe self-managing behaviours is proposed in an architecture with some more powerful nodes being in both the WSN and in the P2P overlay. It does not, however, consider large scale deployment, energy efficiency, discovery or provide detailed design or implementation or results.

3.8.5 TinyTorrents

TinyTorrents is an interesting example of an approach where a gateway is used to bridge nodes in a WSN to nodes in a wider network. In this case, the WSN nodes are represented as peers in the BitTorrent network. TinyTorrents aims to provide "reliable, redundant and distributed dissemination of WSN data across the WSN and Internet" [118]. It allows end-user applications to access, consume and aggregate data from multiple WSN sources. A gateway node runs the TinyTorrents software and a BitTorrent client such as Vuze [119]. This gateway can then communicate with a WSN node using TinyTorrents and communicate with peers in the BitTorrent network using BitTorrent. The integration with Vuze allows remote BitTorrent clients to access the data in the WSN.

It uses its own specific routing layer (TinyHop) to provide reliability in the wireless network and to reduce energy use. TinyHop assumes that any node can act as an on-demand sink, where a sink is a gateway that can integrate with a gateway to the Internet. This is done in order to share the traffic load and avoid excessive load on nodes that may result in those nodes running out of power and so cause network partitions. TinyHop uses flooding, but with specific measures to reduce overhead and prevent cycles.

The TinyTorrents P2P protocol is used to communicate between two peers. TinyTorrents uses concepts from BitTorrent for the transfer of files and it retains the use of a torrent file. A torrent descriptor holds the number of pieces, the checksums for each piece, the unique key for the torrent and the length of the file

to be transferred. This key may indicate attributes such as the type of data (e.g. temperature), rather than a straight SHA1 hash as in BitTorrent. Application agents give access to the torrents in the WSN, e.g. those related to temperature readings, and may filter or aggregate the data returned from the WSN. A TinyTorrents peer holds a bit vector to indicate the file pieces it holds and exchanges this with other peers rather than exchanging messages for each piece it has. TinyTorrents uses a tracker node in the WSN, which holds the address of each peer in a particular torrent and a node can get this peer list from the tracker. Testing showed that the node doing the initial seeding of the file bears most of the load and a strategy was necessary to adjust a node's seeding duty, e.g. to maximise battery life [118]. Tests also showed the load on the tracker node was relatively insignificant, but the cost of the initial flooding was significant. It was also found that caching file pieces on intermediate nodes on a route reduced the number of messages sent, with potential benefits in reducing power consumption on nodes.

It can be said that TinyTorrents' findings on caching data and its use of a torrent approach are interesting, but it uses file pieces rather than a more natural query for sensor data. Furthermore, it shows the potential of a full integration of a WSN with an external application layer as part of an overlay network, albeit it is limited to BitTorrent. Its use of a centralised tracker proved, however, to be problematic. They did not pursue a DHT based tracker, believing it to be inefficient in sensor networks based on [120]. That paper, however, focuses on wireless ad-hoc networks and presents a DHT using clusters of nodes supported by position based routing, where it uses the hash to determine the cluster to hold a value (unlike the WSN case where the data starts on a source node). Furthermore, while Kademlia existed at the time of the TinyTorrents paper, the possible use of Kademlia as a trackerless torrent in BitTorrent is not shown.

3.9 Cache Algorithms

Caches are used to improve performance by serving data from faster local memory rather than from slower disk memory in the case of OS paging or from remote servers in the case of Web Caching (and proxy caches). Given the volume of sensor data, the limits on bandwidth and possible constraints on response time, in IoT in general and the cost of sending data over a wireless network in a WSN

in particular, the use of caches may be appropriate and this section considers a number of cache techniques.

Important considerations in the design of caching algorithms are the nature of what is being cached, i.e. how frequently is it updated, what size it is, what consistency is required, where to cache and when the data is to be removed. Strong cache consistency is usually required in paging or in a cache supporting a database. Web caches generally use weak consistency to reduce the network related overheads. This is often realised using a TimeToLive (TTL) set using the HTTP Expires header and where a request for a document in the cache longer than the TTL results in a HTTP IfModifiedSince GET request.

The data to be removed is determined by the cache replacement policy, which aims to maximise the use of available resources. The success of a cache algorithm in a given scenario is generally measured by the hit ratio, i.e. how many requests are served from cache. In a sensor node, the amount of memory will be constrained, so the cache replacement policy is important in terms of managing the cache to achieve a reasonable hit ratio, without requiring a large overhead in code size or processing time. Three main approaches for cache replacement are used [121]:

- Applications provide future access hints, e.g. based on a query to be performed.
- Explicit detection of access patterns unfriendly to cache algorithms like Least Recently Used (LRU) and which results in a switch to other replacement strategies.
- Tracing and history of accesses, which are only useful if past accesses are likely to be reflected in future accesses.

Rather than selecting entries to remove on some access history, alternative strategies include Random Replacement (RR) or expiring cache items after a time. An example of expiring cache items can be seen with Redis [78], which is used as an in-memory cache. In Redis 2.x, a key is actively expired when a client tries to access it and the key has timed out. For expired keys that will not be accessed, it periodically(10s) tests 100 random keys with an expiry time. All the expired keys are deleted from the keyspace and it selects another set of random keys if more

than 25 keys were expired, continuing until less than 25 expired keys are found. This assumes that these random samples represent the whole key space.

The following sections consider use-based cache, web cache and paging cache replacement algorithms.

3.9.1 Use Based Cache Algorithms (LRU, LFU, MRU)

The Least Recently Used algorithm replaces the LRU item, i.e. the one which has not been accessed for the longest time. It uses a list of items ordered by their last access time, with the LRU item at the bottom. Its simplicity makes it widely used, although there is a cost to maintaining the order on every access. It is suitable for workloads showing locality, where an item is accessed shortly after a previous access, i.e. there is a small reuse distance. It does not distinguish a recently added (and never accessed) cache entry from one accessed frequently but not recently. This reduces its effectiveness when there is less locality, e.g. a scan accessing a series of items once only has no hits, but flushes other pages which may have been accessed again.

To improve its effectiveness, LRU/k gives priority to items based on their kth most recent access, i.e. the normal LRU is LRU/1 using only the most recent access. LRU/k does, however, require log N work (for N pages) to manage a priority queue for each page access. Another LRU variant is Pseudo-LRU (PLRU) used in CPU caches with large associativity where LRU's implementation cost is too high. 2Q [122] is another of the LRU family of algorithms and was designed to provide constant overhead per access as in LRU, while achieving a similar page replacement performance to LRU/k. 2Q places a page (or its reference) in the A1 FIFO queue on its first access. If the page is accessed while on A1, it is termed hot and moved to the main Am queue, which is LRU. If it is not accessed while on Am, it is considered cold and removed.

The Most Recently Used (MRU) algorithm removes the most recently used items first and as such it is most useful when the older an item is, the more likely it is to be accessed, e.g. it outperforms LRU for random access or repeated scan patterns.

In contrast to LRU, the Least-Frequently Used (LFU) algorithm increments a reference for count on access and removes first those items with the lowest count,

i.e. used least often. It is not efficient to implement with a single linked list as removal or insertion would be O(n) and has the disadvantage that a previously frequently accessed item may remain in the cache a long time and not be replaced by more recent items.

3.9.2 Web Cache Approaches

Web caches and proxies are concerned with managing caches in order to ensure a good hit ratio and to achieve reduced access latency and better bandwidth usage. LFU-Aging extends LFU with the recency of last access of a cached document to overcome the LFU issue above. LFU with Dynamic Aging (LFUDA) uses a dynamic aging policy, where a cache age factor (less than or equal to minimum value in the cache) is added to the reference count when a document is added to or updated in the cache to avoid documents that were previously popular and have a high count staying in the cache too long [123].

The Greedy Dual-Size (GDS) policy takes into account size and a cost function associated with fetching objects [124]. The GDS-Frequency variant of the Greedy Dual-Size includes frequency of reference in a key for more popular, smaller items. Its key also includes the size and the cache age factor to handle previously popular documents to reduce the LFUDA issue above.

The Squid web proxy [125] uses an LRU list with a scheduled process to periodically remove objects. It also keeps cache disk usage between low and high water marks with a LRU threshold adjusted according to the cache size proximity to a watermark, i.e. it measures how long it would take to fill the cache at the current request rate. It also uses GDS-F and LFUDA.

3.9.3 Paging Algorithms

Paging algorithms tend to be simple to reduce the overhead so as to not adversely affect performance. Their relative simplicity, low overhead and small code size make them worth consideration for use in resource constrained nodes.

3.9.3.1 CLOCK Algorithms

The original CLOCK algorithm [126] uses a circular list of fixed sized pages to avoid moving data when re-arranging the list. A page reference bit is set when a page in the list is accessed. This circular list can be considered as a clock, where a hand of the clock points to the oldest page in the list, as shown in Figure 12.

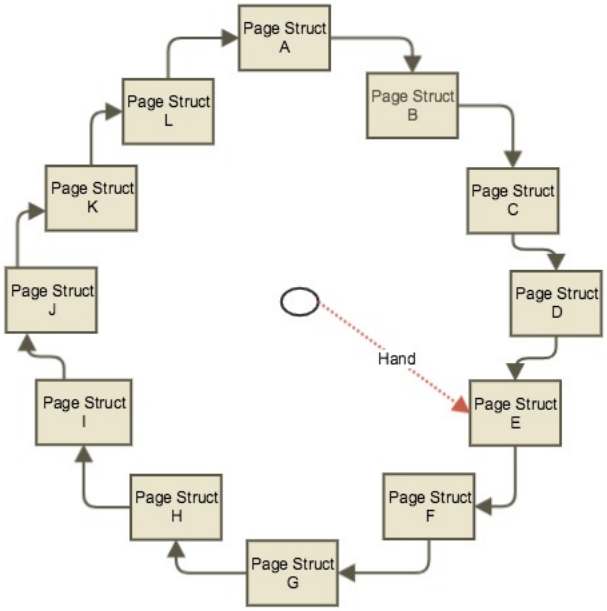

Figure 12 Clock Algorithm

On a page fault, the reference bit of the page at the clock hand is checked. If it is not set, the page is replaced by the faulting page, otherwise it is reset and the hand moves through the list clearing page reference bits until it finds a page reference bit of zero and replaces that page. The clock algorithm approximates LRU and avoids its ordering of the list, but shares its lack of scan resistance.

The WSClock Algorithm [127] is a variant where the task's virtual time and a page's last reference time are used as part of determining if the page should be replaced. GCLOCK [128] has a counter per page which is incremented if a page is hit and the clock hand sweeps the pages decrementing the counters until selecting a page with a zero count to be replaced. Linux used page aging, where the sweep increased the page age by a constant if its reference bit is set and decreased it by a

constant if not, but setting these values has proven problematic across workloads. CLOCK-Pro [129] is another variant that manages a list on page faults and also keeps track of a limited number of replaced pages to overcome the LRU problems with scan and loop. CLOCK-Pro uses the replacement policy principle from LIRS (Low Inter-reference Recency Set) [130] of replacing a page with a high reuse distance (the time in terms of the number of other distinct pages accessed since its last access) even if it is recent. A single list of pages is ordered so small recencies are at the head and large recencies at the tail. It uses three hands to sweep the list: one pointing to the list tail (the last hot page), one pointing to the last cold page and a third pointing to the last cold page in a test period. Cold pages remain in the list for a test period, becoming hot if accessed during that period. The test period is the largest recency of the hot pages, unlike the constant threshold used in 2Q and CAR to distinguish hot and cold pages.

Clock with Adaptive Replacement (CAR) [131] is self-tuning and incorporates frequency with two clock lists, where T1 contains pages with “recency” and T2 contains pages with “frequency”. New pages are inserted into T1 and move to T2 based on a test of long-term utility or frequency. It also uses a list with history of recently evicted pages from T1 and T2 to adaptively determine the list sizes. It has a cost close to CLOCK, but with similar performance to ARC and better scan resistance than LRU.

3.9.3.2 ARC

ARC [132] uses two variably-sized lists (with a combined size of twice the number of pages) to hold the history access information for referenced pages. One list holds cold pages (touched once recently) and the other holds hot pages (touched at least twice recently). The key idea is that the space allocated to the pages in each list is managed based on which list had the most recent misses using a ratio of cold/hot page accesses. It does not handle the locality of pages in the two lists, so a page that is regularly accessed with a reuse distance a little more than the memory size may get no hits.

3.9.3.3 MultiQueue (MQ)

MQ [133] has *m* lru queues, where queue *i* contains pages that have been seen between *2i* times and *2i-1* times recently. It increments the page frequency on a hit and the page becomes the MRU in the relevant queue with an expire time in that queue using a lifetime parameter which is tuned based on a temporal distribution. On each access, the expire time is checked for the LRU in each queue and it is moved to the next queue's MRU if expired. Hence, MQ has constant-time overhead, but this check for *m* queues per request means it has higher overhead than LRU, ARC, and 2Q

3.9.4 Summary of Cache Algorithm Performance

LRFU [121], LRU-K [134] and MQ [133] have been found to be more expensive than LRU, while LIRS [130] and ARC [132] have a cost similar to LRU. 2Q [122] has been shown to be very sensitive to its parameters and may perform much worse than LRU. CAR has a cost close to CLOCK, but with similar performance to ARC and better scan resistance than LRU.

3.9.5 Caching in Wireless Sensor Networks

Extending node battery life is often addressed by managing node duty cycles, but the use of caching data on nodes in WSNs has the potential to reduce energy use across nodes by reducing transmissions (and hops) and also to support local data analysis. Adding a cache to WSN nodes is, however, problematic owing to the following factors: the limited node memory to hold cached data, the limited node processing power, the limited period for which data is useful and the lack of a system architecture that easily incorporates cached data. Furthermore, returning cached data from a node closer to the requesting node than the source node will reduce the number of transmissions required and so reduces interference effects. It will also reduce the response time, particularly when requests from multiple nodes are answered using the same cached data.

CoAP supports a simple cache in an endpoint or an intermediary, using freshness and validity information in the CoAP responses. The cache allows an earlier response message or a stored response for the current request. The origin server provides an expiration time using the Max-Age Option and an ETag Option in the

GET request allows an origin server select a stored response to use and to update its freshness.

Caching has also been investigated for WSNs in the context of co-operative caching to serve data with low latency and to reduce energy consumption, where each node constructs responses to queries by cooperating with its neighbours, but with little focus on the cache replacement algorithm itself. A key aspect of this co-operative approach is to identify which nodes will implement the co-operative caching decisions, e.g. which node makes forwarding decisions or which nodes get the requests for data [135]. Some approaches calculate a Node Importance Index, requiring nodes to hold their neighbours' connectivity state and so lack robustness [136].

Data replication and caching strategies have been considered in Mobile Ad-hoc Networks (MANETs) [137], but some schemes [138] require knowledge of network topology and involve periodically moving data, both of which would reduce their effectiveness in a WSN, especially if data access patterns vary or nodes join/leave. Static approaches to cache placement [139] are similarly limited in the WSN scenario. COOP [140] keeps a table of previous requests and the nearest relevant cache, using flooding to find the data only in the case of a miss. The hybrid cache for cooperative caching in MANETs [141] does not require the selection of special nodes and shares data only on the path between source and requester. When forwarding data, a node may cache either the data or the path according to the data size, TTL and number of hops to be saved.

In Directed diffusion [74], intermediate nodes may cache recently sent data messages or aggregate data, although the emphasis is on the routing and filtering of data and the matching of interests rather than cache implementations.

3.10 Summary

This chapter has outlined the potential of Big Data and Cloud services to handle the volume of IoT data and to be able to scale to allow a range of universally accessible services to use that data. This is particularly required to support the emergence of fog and edge computing to address bandwidth and latency issues with the use of Cloud services in certain scenarios. This chapter has shown that

this potential is, however, only being partially realised with isolated, independent solutions and limited interoperability or orchestration.

This chapter presented an overview of a number of approaches to distributed systems that have been shown to provide interoperability and orchestration, such as tuple spaces, including research on the use of tuple spaces in WSNs indicating their suitability as a distributed system to use in IoT, particularly its use of a few simple primitives and decoupling. It also presented P2P, particularly DHTs, in some detail to illustrate the range of approaches and the scale achieved. It has also shown that previous approaches to using P2P in WSN range from considering a more powerful node in the WSN as a peer in a wider network to the approach of a P2P overlay network, including all the sensor nodes in a WSN. It also discussed other approaches such as distributed databases, directed diffusion and their limitations, e.g. their limited flexibility in terms of scenarios they were designed for.

Similarly, a number of cache algorithms were presented in detail due to the potential benefits of caching in IoT, while recognising the difficulties of including caching in a constrained WSN device.

The next chapter analyses the research outlined here and presents a set of requirements for the type of interoperable and scalable IoT that includes WSNs as discussed in chapter 2. It then explains the design choices made using those requirements and uses that analysis to present a holistic architecture for IoT.

4 Analysis, Architecture and Design

The expected growth in the number of devices in IoT outlined in earlier chapters presents the challenge of creating an architecture able to scale the technology down to resource-constrained WSN nodes and to scale up to an overall IoT with many billions of devices [10]. This will require seamless interoperability at the device, network and service levels with sets of abstractions. Such seamless interoperability will allow IoT to become a realisation of Mark Weiser's vision of ubiquitous computing where tiny networked computers become woven into everyday life [11] and in which WSNs will become extensions of the IoT infrastructure. It would also be in line with the vision for fog and edge computing where they take full advantage of Cloud and Big Data to allow sensor data and services to be universally available.

One view [142] considers that systems using applications that make use of resources (storage, cycles, content, human presence) available at the edges of the Internet is P2P if it meets the test "Does it treat variable connectivity and temporal network addresses as the norm?" and "Does it give the nodes at the edges of the network significant autonomy?". This definition and its inclusion of the role of edge nodes makes this broad view of P2P relevant in the fog computing scenario, e.g. Figure 8 from the OpenFog Consortium [95] illustrates the diverse range of devices, services and roles from the edge to the Cloud.

To address this challenge, this chapter presents a set of architectural requirements and an analysis of the prior research presented in earlier chapters. The variety of sensor, user applications and network scenarios for IoT and WSNs as outlined previously, require the communication protocols and abstractions to be flexible and able to support a range of sensor types and application requirements. These application requirements range from infrequent scheduled requests for limited amounts of data to frequent (and possibly asynchronous) requests for large amounts of data and for actuation functionality on those devices.

This chapter then describes the resulting layered architecture and abstractions for the roles taken by services on (WSN) nodes and in the Cloud, supported by a new P2P protocol, termed the Holistic Peer to Peer (HPP) protocol. This approach uses data sharing and co-ordination concepts from tuple spaces, P2P protocols (particularly Freenet and JXTA), and the Kademlia DHT as well as a novel caching algorithm, called CacheL, for resource constrained devices.

4.1 Analysis

4.1.1 Architectural Lessons

A key lesson that can be learned from the success of the RESTFul architectural style is that consistent abstractions are required to simplify the development and deployment of nodes and applications in order to achieve an interoperable architecture. The layered Eclipse and OpenFog architectures present a valuable consideration of the range of actors and scenarios in IoT now and in the future, but they distinguish between the different entities with separate abstractions and do not provide a consistent set of constraints (as in the RESTFul approach) or abstractions. Similarly, they do not specifically consider entities as having a peer relationship as BitTorrent does. This leads to the need for edge proxies with different architectures and abstractions, reducing the ability to achieve large scale [67].

The RESTFul Architectural Style has proven its scalability and flexibility. Given scale, flexibility and interoperability are desired in IoT, the work in this chapter is based on a similar architecturally driven approach, beginning with a set of abstractions for a holistic architecture and protocol. Furthermore, the scale and autonomy possible with P2P, as demonstrated by BitTorrent, suggest the use of P2P in IoT is appropriate.

This use of an integrated approach to combine a number of concepts to realise the vision of IoT has been expressed elsewhere as “algorithms ...are becoming key, but above and below the “system waist” represented by middleware. Below it, algorithms for optimizing low-level system concerns (e.g., power consumption), possibly in concert with the application goals, are more and more important….. Above it, algorithms for automatically mining and exploiting the wealth of “big data” harvested by mobile users are rapidly becoming one of the most exciting challenges of future computing” [143]

4.1.2 Architectural Requirements

The objective of the proposed holistic architecture is to simplify the development, configuration, scalability and deployment issues of WSNs. This will then provide an environment that enables a wider deployment of WSNs and easier interfacing to other networks, while also providing consistent abstractions to enable the easier development of generic and more powerful applications to take advantage of the sensor data.

To meet this objective and derived from analysis of applications discussed in section 2.7 and the analysis of research presented in earlier chapters, the following requirements are defined for a holistic architecture:

- **Req-1**. It must clearly define the possible roles of nodes. It is unreasonable to demand that all nodes have equal functionality, as this limits the ability to handle more powerful nodes. Nodes will, however, require a minimum level of functionality, e.g. responding to a request for its capabilities or forwarding data to a neighbour.
- **Req-2**. It must provide abstractions for the basic operations required of a sensor node and the services using it, which map easily to a range of heterogeneous devices and higher level services.
- **Req-3**. It must be independent of particular node hardware and must handle a range of node functional capabilities.
- **Req-4**. It must provide simple, consistent APIs for developers of device and application software.
- **Req-5**. It must provide a consistent means to exchange sensor information independent of the underlying technology and provide specific support for the modelling of sensor data to allow integration into higher level systems, be it fog or cloud based, possibly using RESTFul interfaces such as the existing OMA LWM2M.
- **Req-6**. It should support a sensor node informing other nodes and services of its sensing and platform capabilities.
- **Req-7**. It must be able to handle small, static networks and allow the system to adapt as the network grows/changes due to mobility (as users or their nodes join/leave) or encounters other networks, supporting applications discovering and collaborating without a centralized

coordination facility. As pointed out earlier, the key to such a network is the ability to identify and to route to peers, regardless of the underlying layers or network (WSN or otherwise)**.**

- **Req-8**. It must use protocols that are sufficiently simple for low capability devices to participate, according to their capabilities.

The need for such a holistic approach can be seen in a remote healthcare monitoring scenario, where sensors connect to a central gateway in a house over a wireless network. The gateway is responsible for storing the data locally and uploading data to a central health monitoring site, possibly via a central gateway/proxy and cloud based services to analyse the data [144]. Such solutions often require sensor application and proxy design to handle data integration, network integration and security concerns. This lack of unified abstractions will become more problematic when healthcare scenarios such as Wireless Body Area Networks are deployed, e.g. IEEE802.15.6 allows up to 64 nodes on a body in a star network via a central co-ordinator node (which will connect to an external gateway).

An important part of the holistic approach is to eliminate specialised edge nodes/sinks and proxies as this can be considered to be important in achieving really large scale [67]. It will be required to integrate with other systems, such as OMA LWM2M and fog components as per **Req-5**, but this should be achieved in a more consistent manner by having a holistic approach.

These requirements can be compared to the common principles described for WoT in section 2.6 as follows, although WoT is specific to Web technologies, unlike Req-1 to Req-8:

- **Req-2** is a more general statement in terms of abstractions than in the WoT statements that the WoT architecture should enable mutual interworking of different eco-systems using web technology, as well as that the WoT architecture must enable different device architectures and must not force a client or server implementation of system components.
- **Req-4** and **Req-2** are more general than the WoT statement that the WoT architecture should be based on the web architecture using RESTful APIs

and the WoT statement that the WoT architecture should allow to use multiple payload formats which are commonly used in the web.

- **Req-5** is similar to the WoT statement that the WoT must provide interoperability across device and cloud manufacturers.
- **Req-7** is similar to the WoT statement that the WoT architecture should be able to be mapped to and cover all of the variations of physical device configurations for WoT implementations. The WoT statement that the WoT must be able to scale for IoT solutions that incorporate thousands to millions of devices is more specific than **Req-7** regarding scale.
  WoT is more specific in stating that it should provide a bridge between existing and developing IoT solutions and Web technology based on WoT concepts and that WoT should be upwards compatible with existing IoT solutions and current standards.
- **Req-1 and Req-6** regarding roles and the discovery of node capabilities are not specifically covered in the WoT common principles, but two roles are called out in the WoT abstract architecture – thing and consumer. The functional requirements do also cater for device descriptions and for discovery mechanisms based on the Thing Description as outlined in section 2.6.
- **Req-8** specifying the support of constrained devices is not specifically covered in the WoT common principles, but constrained devices are considered in some places, e.g. the use of a digital twin to run a less power constrained device or the use of a directory to hold Thing Descriptions (TDs).

The comparison above of the requirements defined for a holistic architecture and those set for WoT indicate a shared view of the importance of interoperability and scalability. It also shows that the requirements for the holistic architecture are sufficient and more general, except when being specific regarding constrained devices, whereas those for WoT are very specific to the use of Web technologies and bridging.

4.1.3 WSN Software Development Requirements

Programming WSN applications and nodes is time-consuming, error-prone and difficult requiring a high level of knowledge about low level hardware and

network technologies. This is usually done using a vendor specific environment for particular hardware, with a set of tools and libraries for handling the low level aspects. While acceptable in specific domains, this limits the integration of WSNs into the type of the seamless, context aware environments envisaged in pervasive computing and IoT, where applications/services are interested in the sensed information itself and not the underlying hardware or wireless network.

Software Engineering concepts, new methodologies and abstractions to improve the development process and ease the integration with other systems are required in order for WSNs to be more widely deployed [143]. These would allow developers to move from the current code-and-fix process, which is reliant on the primitive constructs provided by the types of operating system and environments described in chapter 2 and also on the skills of developers.

In terms of the architectural requirements outlined above, **Req-2, Req-4 and Req-5** will provide the higher layer abstractions and consistency to make the software development process less reliant on the specifics of a given wireless network, hardware device, OS or development environment. **Req-1, Req-3, Req-6, Req-7, Req-8** will then allow the software developed to handle the heterogeneity in a range of scenarios and so achieve the interoperability desired in IoT. This will help developers to more easily create software for multiple platforms, without having to learn many different abstractions and APIs and also remove the need to maintain separate versions of software for each device platform and service.

4.1.4 System Model

Based on the earlier overview of the types of WSNs and nodes, a broad model for a WSN and IoT is assumed. This model consists of sensor nodes using bi-directional links in a multi-hop manner and nodes in a WSN being able to communicate over the Internet with more powerful nodes and services in the Cloud. No assumptions are made about the size of the network. Elements in the proposed holistic architecture, such as node message transfers, cache algorithm, tuple-space store, must not impose undue communication overhead, e.g. the cache algorithm does not require flooding and can handle the dynamic nature of the WSN in terms of the source and destination of requests. No assumption is made regarding use of a static topology and there is no requirement for data relocation

or recalculation to update topology related data when nodes join/leave. It is assumed that nodes may have different capabilities; some nodes will be able to cache data or issue node identifiers and other less capable nodes will only act as sources (or sinks) or forwarders of data and more capable nodes will cache or aggregate data. Data will be retrieved from the source node in the absence of a cache. No assumptions are made about the routing algorithm used and the only assumption is that nodes respond to messages according to their capabilities and that data may be cached as it is forwarded, preferably peer to peer. While such forwarding may be as a response to a request, data may also be pushed from nodes.

4.1.5 Security Considerations

Security will be based on the requirements of the particular application and where it is deployed, similar to the approach taken by JXTA. The HPP protocol will not require the use of a specific authentication, security or encryption technology. The format of HPP messages will, however, enable the carrying of metadata information, such as digests or certificates or credentials, e.g. in the form of an opaque token that may be required in every message. HPP messages may also be encrypted and signed for confidentiality and refutability, but this will use whatever methods are appropriate for the particular scenario and may use security in the lower layers of the stack as per 2.4.1. This security may be per device or per user on that device, as required by the node(s) determining access.

Furthermore, and in line with the decentralised model of P2P, this use of security will be controlled by the nodes and the groups themselves. The required security will be determined during the initial exchange of messages as per 4.5.5.1, although this will require new work to define the management policies and their distribution to the (usually bootstrap) peers. This checking of security at the initial exchange will enable supporting the mobility of devices and users.

The use of roles within the holistic architecture (and the exchange of device capabilities) also provides architectural support for specific security considerations, e.g. roles could be defined and shared for specific security purposes such as those foreseen in the more powerful nodes at the edge of the network in fog computing. A further point of interest is that a P2P architecture

means that nodes can only make outbound connections, which may be required in certain security scenarios.

4.1.6 Findings from the Review of Prior Work

Earlier chapters have shown that many approaches have been investigated and implemented to support WSNs, ranging from protocols such as CoAP and MQTT to Middleware solutions to Cloud specific SDKs, with a range of data storage and dissemination approaches. Those chapters have also shown that there is an absence of broad, simple application layer abstractions and so applications are often bound to a particular WSN technology, e.g. the use of a Zigbee device combined with use of a specific Zigbee profile for an application area. In particular, a number of approaches have separated the WSN and the rest of the IoT due to the expected limitations of the constrained WSN devices, e.g. those approaches outlined in section 3.8.4, which is against **Req-2** and **Req-5** above.

Earlier chapters have also shown that many current middleware platforms focus on abstracting the device and sensor functionality to achieve interoperability at the level of information models, as in OMA LWM2M. There is less emphasis on creating a seamless device to application environment for IoT or the orchestration or co-ordination required in this distributed environment. This has resulted in ad-hoc solutions often based on a Cloud integration for a particular service. The following sections consider the particular aspects of prior work presented in chapters 3 and 4, which are relevant to the design of the proposed holistic architecture.

4.1.6.1 Information Models

Section 2.5 showed that while OMA LWM2M [53] provides solutions for end to end interoperability across different networks and devices, it only offers limited higher level service abstractions beyond client/server. It also showed that the rate at which sensor data changes, or is available, may not be that frequent and is known by the originating sensor. If this information was advertised, it could be used by the application to match its requests to this rate and also to decide whether it is useful to cache this data (and how long to cache it) at other nodes, perhaps closer to the requesting application. Section 2.5 also showed that the Common Information Model (CIM) models sensors well, but does so verbosely

with a rich information model. As such it is a useful information model to demonstrate (or not) the usefulness of the HPP abstractions, even if it is not specific to WSNs.

Thus, the functionality required by a management application is the same in a WSN as in a more traditional sensor application and the use of such information models is appropriate in a WSN. Hence, the emphasis should not be on redefining the use of sensors or the sensors themselves, but in doing so as efficiently as possible given the WSN constraints and in a way which can map easily to formats used by the higher level applications.

4.1.6.2 Data Distribution

Chapter 3 showed that tuple spaces enable a simple and consistent API for a distributed system, possibly including the use of objects as per Javaspaces, which also provides for leases in the tuple space to simplify its management. Section 3.6 showed TeenyLIME and LIME as example uses of tuple spaces in WSNs providing benefits in terms of simplicity and decoupling. TeenyLIME could run on constrained devices, but only a node's local tuple space is shared with the nodes within communication range. LIME extended the local tuple space into a federated tuple space into which tuples can be added/removed, but only when the nodes are in range of each other. LIME was implemented in Java meaning it could not run on a constrained node. Directed diffusion described in section 3.2.2 provided an interesting approach, particularly the routing of data and the expression of interest and use of an interest cache. It is, however, tightly coupled to a query on demand data model where applications can accept aggregated data and this limits its applicability to those scenarios.

Middleware approaches such as Sensation allowed WSNs to collaborate, but the approach of developing a proxy for each network and the requirement for a priori configuration of network profiles (to conceal the underlying heterogeneity of WSNs) limit their use for a seamless, interoperable architecture. Also, agent based middleware approaches that use a set of services and a language to compose sensing tasks from the services are suitable for monitoring moving and dynamic phenomena, but require particular node computational capability due to their complexity and the code mobility reduces node lifetime due to the energy consumption of the additional network traffic.

Based on this, the use of tuple spaces with a lease to help its management can be used as the basis of a data dissemination approach that provides both a simple and consistent API and can provide the required decoupling for a distributed system. The challenge is to ensure that this can be implemented on a constrained system without undue overhead and to ensure that it can make the development of software easier.

4.1.6.3 Caches in WSNs

The limited available storage on nodes means that only limited amounts of data can be held on nodes, requiring careful cache management, while the limited processing power and memory for code mandates the use of a simple cache management algorithm. The fact that local analysis of node data is often most useful for a recent time period also suggests that caching is appropriate in WSNs, particularly given the move towards fog computing. Facilities for caching are included in CoAP, particularly for more powerful nodes, and caching was considered in directed diffusion for interests and co-operative caching has been investigated, but caching has not been considered more fully as a part of an architecture across a range of node capabilities in the WSN and beyond it.

This thesis proposes that the key requirements for a cache on a WSN node are:

- Simple to implement and efficient in CPU and memory use.
- Use of a single abstraction for data storage. There should not be separate stores for the node's own data, remote node data and lease management. This is in order to reduce the code size and also to provide consistency to developers.
- A lease associated with cache data. The lease is set and renewed by the source node as the data may only be useful for a limited period, e.g. only the latest copy may need to be held by a node. Permanently stored data will have an infinite lease.

Use of a lease allows the cache to be used as an intrinsic part of a holistic architecture, because a lease can provide features desirable in a distributed system, such as self-management. For example, the leases for cached data from nodes that have left the network will be removed from the cache on lease expiry.

Furthermore, that cached data may be any node data, including sensor readings or DHT node information.

In order to meet these requirements, this thesis includes the design of the CacheL algorithm for constrained nodes which uses leases as part of its cache replacement policy. From the cache algorithms presented earlier, the Clock paging algorithm was the inspiration for CacheL based on its relative simplicity making it easy to extend and adapt, i.e. it does not require the use of sort (like LRU) and does not require additional communication between nodes, as in several cooperative caches.

The CacheL algorithm is a simple cache algorithm that can provide an acceptable hit ratio, within the limited memory for code and data on constrained nodes. As well as allowing self-management, its use of a lease also provides the capability to represent the validity of the sensed data, as the lease can be set or renewed based on the time that the next reading will be made available.

4.1.6.4 Peer to Peer

Chapter 3 outlined the self-organising, decentralised and distributed nature of P2P systems. The self-organising characteristic allows easier management and deployment, particularly in dynamic or collaborative scenarios where nodes join or leave due to mobility or variations in link quality. IEEE TG6 for Body Area Networks identified P2P as useful for gaming and social networking applications with their dynamic and collaboration requirements [64]. The decentralised characteristic of P2P systems makes them potentially robust to faults and scalable to large numbers of nodes. The distributed nature of P2P means that peers can be physically and logically distributed reflecting the reality of the distributed IoT environment and providing the option of data redundancy by storing in more than one node or shared processing (as required in fog and edge computing). Section 3.8 showed examples of P2P in WSNs and the possible benefits of P2P's self-organisation, decentralisation, distribution and caching. The example in 3.8.1 showed the use of a gateway as a sink for sensors into a wider mobile network [112] and showed the value of abstractions and supporting layers to model sensor characteristics in a WSN. It also showed that the use of a sink to interface to an external network limits scalability and deployment, unless all the gateways shared

networking and sensor abstractions. This suggests the use of a more decentralised P2P approach and that the potential of a pure decentralised, loosely structured self-organising P2P network, as exhibited by Freenet [145], has not been realised. Chapter 3 also showed that Freenet provided "implicit" caching as responses are cached and this potential to reduce network traffic would be valuable in a WSN.

P2P systems may also consist of nodes that are used to form a content/service network, termed an application-level overlay. This aligns with the desired nature of IoT and WSNs described earlier. A P2P overlay network potentially allows applications running on top of various other network and data link layers to not require any knowledge of the underlying implementation, identifying and routing to peers, regardless of the underlying network. This is required as per **Req-5** and in order to reduce the difficulty of developing software for constrained WSN nodes, particularly in the context of the changing and diverse range of devices, services and lower layer protocols. Also, note the earlier discussion of RPL suggested that a single routing standard is unlikely to be able to handle the IoT's expected range of device heterogeneity and application requirements [45]. Coupled with the range of information models, new roles, services and devices expected from fog computing, this means that IoT requires a solution not just at the routing layer. For this reason, an architecture above the routing layer using a P2P overlay is proposed.

4.1.6.5 Distributed Hash Tables

The main purpose of sensor networks is to collect data about some phenomena of interest (and sometimes to actuate a device), with requirements on latency and lookup time depending on the application. As per section 3.7.3, the lookup times achievable by DHTs suggest that the use of such techniques may be appropriate in sensor networks. While section 3.8.2 has countered arguments against the use of DHTs in WSNs, it is important to recognise the potential issues in using a DHT as the system scales, e.g. the cost of maintaining the DHT in a dynamic network as nodes join/leave can become a performance issue. Examples of P2P systems using DHTs include Chord [108] and Pastry [109], which differ in how they build and maintain their routing tables as nodes join and leave. They rely on a somewhat fixed topology to assign data to peers and subsequently look up, e.g. Chord uses a

one-dimensional space to assign Ids for both keys and nodes. BitTorrent [12] uses a DHT based on Kademlia [13].

Kademlia's use of a single routing algorithm is different to the approaches such as Chord or Pastry, where one algorithm is used to get close to the desired identifier and a different one is used for last few message hops. Pastry and Tapestry are similar to the XOR symmetric behaviour in their first routing phase, but require secondary routing tables for the subsequent phase with its consequent overhead [13]. Chord does not share the symmetric property and this limits its ability to use information from the queries it receives. As per [13], this also makes Chord's routing tables rigid as each entry in a node's finger table must store the precise node proceeding an interval in the identifier space. It is unlikely, however, that Kademlia's use of parallel queries to k nodes to avoid timeout delays from failed nodes will be appropriate in a WSN.

Kademlia is chosen to be the basis of the DHT for the P2P overlay in the proposed holistic architecture for the following reasons:

- As per **Req-7**, a scalable, robust approach that can cater for both the WSN and wider Internet use case is required. Kademlia has demonstrated scalability and robustness in the case of its use in BitTorrent as a tracker for large numbers of nodes including the handling of nodes joining or leaving.
- Kademlia's use of a single XOR based single routing algorithm makes it simpler to implement than Chord, Pastry and does not require secondary routing tables.
- Kademlia minimises the number of configuration messages required for nodes to find about each other, as this information is also carried in messages used to lookup keys.
- Kademlia's symmetric routing algorithm facilitates the use of caching.
- A node knows much more about closer nodes than distant nodes, as the key space represented grows with the power of 2 with the distance.
- Kademlia nodes can use metrics to route queries through low-latency paths, based on a concept of storing ⟨key, value⟩ pairs on nodes with ids "close" to the key.

The size of the Identifier in Kademlia (20 bytes) may seem large for use in a WSN, but it does allow identification of a WSN node as for all other nodes and also beyond the WSN itself. Furthermore, as pointed out by [113], real deployments often require assigning nodes a globally unique identifier, e.g. to support network management, and so this can be provided by a DHT and not considered an overhead of a DHT. The identifier size can also be reduced in some cases, e.g. by assigning smaller locally-unique identifiers for use locally within a sensor network. Furthermore, DHT computation is within the capabilities of simple node platforms.

It is also worth noting that BitTorrent extended the messages in Kademlia and introduced its own separation of node identifier and info-hash, but that it retained the essentials of the node lookup and associated buckets. It also changed the refresh behaviour and set k to 8 as this was considered sufficient to reduce the probability of that number of nodes disappearing from the network within the refresh periods. This could be used to address the overhead of republishing/refreshing data and the number of buckets, which reduces the overhead of the routing tables and the number of messages exchanged. This could make Kademlia more suitable for WSN use.

4.2 CacheL Algorithm

As discussed in section 4.1.6.3, the successful use of a cache enables reduced communication and so in a WSN could reduce the response time, extend the battery life of WSN nodes and reduce interference effects due to the use of fewer hops to send messages.

CacheL is a low overhead replacement algorithm that performs both lease and cache management. Its novelty lies in its intrinsic use of leases in a cache replacement policy for WSN data inspired by the Clock paging algorithm and its relative simplicity. This means that it does not require the use of sort (like LRU) and does not require additional communication between WSN nodes, as in several cooperative caches. This resulted in the CacheL algorithm being relatively straightforward and suitable for use in constrained WSN nodes.

The introduction of a lease to cache management in WSNs is important as it provides for self-management, e.g. to handle WSN failures where if a node fails then leases on its data will not be renewed so the data will be removed from the cache on expiry, freeing valuable storage space. It also provides the capability to represent the validity of the sensed data, as the lease can be set or renewed based on the time that the next reading will be made available. Incorporating lease management into the replacement process also removes the need for a separate periodic process to manage leases and to expire items. e.g. using time buckets to hold items according to lease times and updating these when the bucket period has passed.

A source node requests a lease from the node it sends its data to, but the lease is granted by the caching node and it can use the size of its cache, the item's size and the lease requested to determine the lease granted, e.g. it may not cache items above a certain size at all or do so for a shorter period than smaller objects. The lease is granted in milliseconds and is not tied to the actual time on the source or caching node. The lease is decremented based on time differences using the time on a node and is renewed by the source node, based on when it received the granted lease response. Hence, there is no need for a global time across the WSN to support the cache lease.

CacheL's commonality of code and metadata for cache and lease management reduces code size and memory use, as well as making the code simpler. Furthermore, allowing the periodic update of leases and expiry of items fits well with the WSN use case of running only when the node is awake.

CacheL is designed for resource constrained nodes to manage items in a node’s datastore as a cache. CacheL extends the Clock algorithm by adding a lease to each item cached and using this lease in its cache replacement policy. CacheL combines application hints (leases set by the source node give a hint when the item can be removed) and history based on access. The lease has priority over access count in determining item replacement, because the source expects an item to be cached for the period granted and treating access count equally would lead to removing entries with long leases. A lease can be renewed by the data source, so that it is not just a TimeToLive on the data. Unlike paging algorithms, CacheL

does not assume a fixed cache size with fixed size buffers and it updates metadata associated with items cached instead of setting the reference bit.

CacheL does not keep lists sorted by access count or time, but replaces the first one or more items it finds with expired leases or which have not been accessed a set number of times (similar to CLOCK-PRO's use of access counts). It manages leases by iterating through items on a specific event, such as adding an item or a time epoch passing. CacheL can also handle items without a lease and evicts them simply based on access count, similarly to the CLOCK algorithm. Items that are accessed frequently, but whose leases have expired can be handled according to a policy, i.e. whether to always remove on expiry or to retain them in the cache if accessed.

CacheL holds the entries to be managed in a list, but unlike LRU this list does not have to be held in any specific order. It also has at least one other list/queue, a pending queue of item references whose lease is due to expire and it also holds their access count. This means that replacement is not just based on count like 2Q. This also has similarities to MQ, but is simpler and done per access as with MQ, but also on a periodic basis. The algorithm sweeps through a main queue until it has found enough items to evict (using count or memory size). This sweep is done when an attempt is made to add to the cache or using the fronthand and backhand time periods, where the fronthand is less than the backhand.

It operates using two sweeps of lists, each performed at their set frequency, as shown in Figure 13. The backhand sweep is primarily used for lease management, but also moves items according to count. This sweep iterates through all the main queue or until it finds one or a specified number of items to remove. It puts items on the pending queue (or a count queue) based on their remaining lease or deletes them if they were not accessed since the previous sweep. This sweep through the main queue decrements the access count, but does not clear it to give some precedence to items with multiple accesses. The previous accessed time is not stored as that is implicitly handled by the periodic sweeps. It holds a reference to the last item checked on a sweep to start the next sweep.

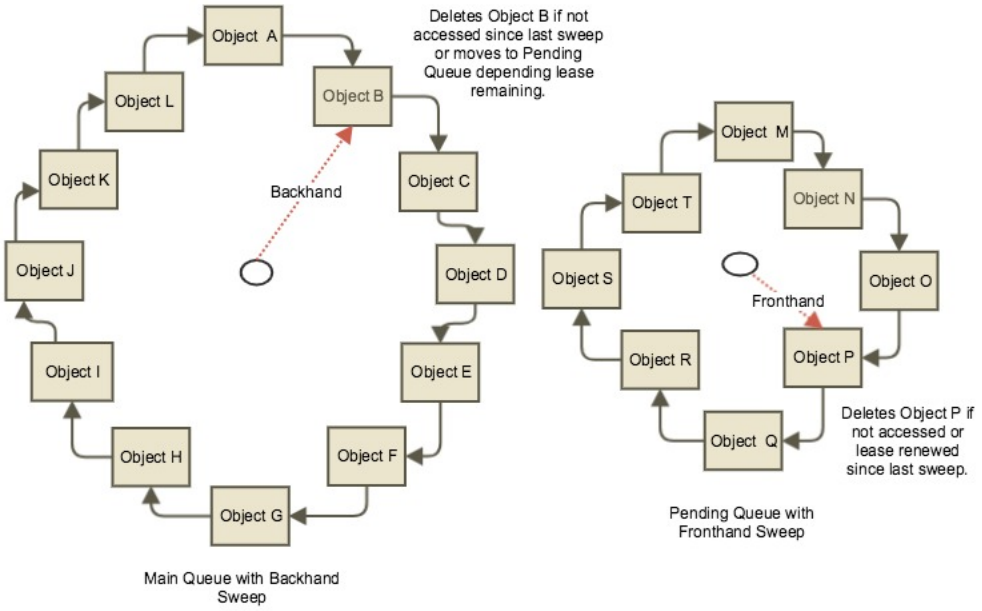

Figure 13 CacheL Algorithm Lists Sweep

The fronthand sweep iterates through the pending queue. Items on this queue are due to expire soon, so this sweep removes them, unless they have been accessed or had their lease renewed, and decrements the access count (effectively setting the time for an item to be accessed before it will be deleted). This pending queue sweep will be called on each add and removes the specified number of entries with expired leases. Items are added to the tail of this queue, but it is searched from the head increasing the time an item can remain on the queue if items expire ahead of it. Items are not held in order of their remaining lease or recency of access as with LRU.

The use of this pending queue means that items with short leases or leases that are about to expire will be removed first, which is reasonable as the source node setting a lease should know how long this data should remain. Items may stay in the cache longer than the lease, however, as the lists are only checked on a sweep to save processing overhead.

A lease threshold is used to select items to put onto the pending queue and by default is set to the time before the next backhand sweep - increasing it may remove items that still have some lease remaining. The period between fronthand

sweeps and a backhand sweep could be tuned using the size of the pending queue, as the time between an item being put on the pending queue on a backhand and its removal on the next fronthand allows for lease renewal (or access).

The following simplified code illustrates the operation of the algorithm. Other code stores the times of the last fronthand and backhand (since_fhand, since_bhand) to determine which sweep to run. Storing the times of the last fronthand and backhand also means the sweep can be run if it is due, when a node that had been asleep wakes up.

```
doSweep() {
 if (since_fhand) //based on time period or counts
    if pending Queue not empty
        deleted = pendQSweep(toDelete)
 // Optimisation added for lots of long leases
 if (min_lease - lease_threshold > since backhand)
     return
 if (since_bhand || // based on time period or counts
     deleted < toDelete))
     deleted = mainQSweep(toDelete)
}
```

The number of items (or size of memory) to free can be set by the caller, rather than just inserting the new page at the selected location on a page fault. It returns the number of entries removed, so the caller can decide whether to allow a new item to be added or to explicitly delete an item, rather than wait for leases to expire. Deleting an item may occur outside cache management, so its reference will either be removed immediately from the pending queue(s) or lazily rely on the lease to expire as it will not be renewed. Similarly, items are either immediately moved from the pending queue on lease renewal (or on access) or lazily on the next sweep.

```
mainQSweep() {
  iterate mainQ starting from last_checked item
  last_checked = this item
  decrement access_cnt
  if (lease set) {
    decrement lease
    adjustLeaseByCnt
    if (lease expired){
      if (access_cnt == 0)
          remove
      else
          move to pendQ
    } else if lease < threshold
          move to pendQ // else stays on mainQ
    } // end of lease expired case
    update min_lease if lease < min_lease
    return
  } // end of lease set case

  if access_cnt <= 0
      remove
  else if access_cnt == 1
      move to pendQ   // or a separate cntQ
  else // if access_cnt > 1
      leave on mainQ, // more precedence than in Clock
}
```

```
pendQSweep() {
 for each item in pendQ
  decrement access_cnt
  if lease set {
      if (lease expired)
          remove
      else if lease > lease_threshold // was renewed
          move to mainQ behind hand pointer
          // has full sweep to be accessed.
      else if (lastAccessTime == 0)
          if access_cnt <= 0,
              remove // not accessed while on pendingQ
          else // (access_cnt > 0 so was accessed)
              move to mainQ // allows lease renewal
    } // end of lease case
 }
```

The following points are noteworthy aspects of CacheL behaviour:

- unlike LRU (and similarly to Kademlia), it is valid for CacheL to not cache an item if no leases have expired, so the sweep could iterate through the entire main queue giving an O(N) worst case. This cost is reduced by holding the minimum lease of an item in the main queue and not doing a sweep if that lease has not expired. Also unlike LRU, a sweep is done on an add or a time period rather than updating ordered lists per access.
- large, infrequently accessed objects with a long lease may remain cached ahead of those accessed more often with short leases. That should be handled by not granting a long lease to large objects when they are added and by increasing the lease based on count during a sweep.
- for items without a lease, the algorithm uses access count only. In this mode, it operates like the CLOCK algorithm, but with the advantage of a pending queue; the main backhand sweep sets a last-checked attribute for a cached item and decrements access count, while the pending queue sweep removes items that have not been accessed. It would be possible to implement a separate count-based list for items without leases or the sweep approach could be easily extended to check other parameters, e.g. the number of hops data has taken. In the no lease case, the algorithm resists scan patterns once the cache is populated, as it adds to the end of the pending queue and the main queue is not sorted.

4.3 Holistic Architecture and Abstractions

The key principle underlying the architectural approach is that all WSNs are primarily about delivering sensed data and events to one or more applications (periodically, on-demand or asynchronously) or delivering commands to actuators from applications.

This principle forms the basis for a more holistic approach allowing WSNs to support a greater variety of users with diverse requirements and a greater variety of WSNs, each with their own underlying technology. The approach is termed as holistic in that it considers the entirety of the flow of data between sensor and service(s), supported by lower layers, rather than being driven by each layer specifying its own behaviour in isolation. The need for such a holistic

architectural approach can be expected to increase as new information models, roles, services and devices emerge from fog computing and other initiatives such as Industrie 4.0. It is worth noting that the Industrie 4.0 initiative considers an end-to-end approach and although it is not a single initiative, a number of shared factors such as interoperability, virtualization, decentralization, real-time capability, service orientation and modularity can be identified [146], all of which (except virtualization) are part of the architecture proposed in this thesis.

The need for such a holistic approach is echoed in [147] by some of the original authors of TeenyLime, where they point out the rich libraries used in mobile and wireless platforms focus on "abstracting the device features, such as on-board sensors and access to the network. Properly managing distributed context, despite the conspicuous literature on the topic, is still central to mobile applications, for which ad hoc solutions are often employed and no prevailing approach has appeared". They attribute this situation to the dominance of the RPC derived invocation-based paradigm, reflected in current service-oriented architectures, "whose tight coupling is intrinsically at odds with the fluid, dynamically-changing context enabled by mobility".

The holistic architecture meets the requirements defined above by using a number of service abstractions to model the different roles a service can perform, defined software layers including an object infrastructure to support information models and a simple protocol, which is based on Peer to Peer (P2P) concepts. In terms of the architecture shown in Figure 14 and detailed in later sections, requirements **Req-1** and **Req2** are provided by the data model service layer**. Req-3** is provided by the local instrumentation layer. **Req-4** is provided by the object space layer. **Req-5** and **Req-6** are provided by the HPP protocol. **Req-7** is provided by the P2P Overlay network built using HPP. **Req-8** is met by all the layers. In terms of the broad software development requirements outlined in section 4.1.3, the approach proposed in this work has adopted a simple set of APIs, abstractions and defined software layers, which both Linda and RESTFul style approaches have shown to be of benefit to developers.

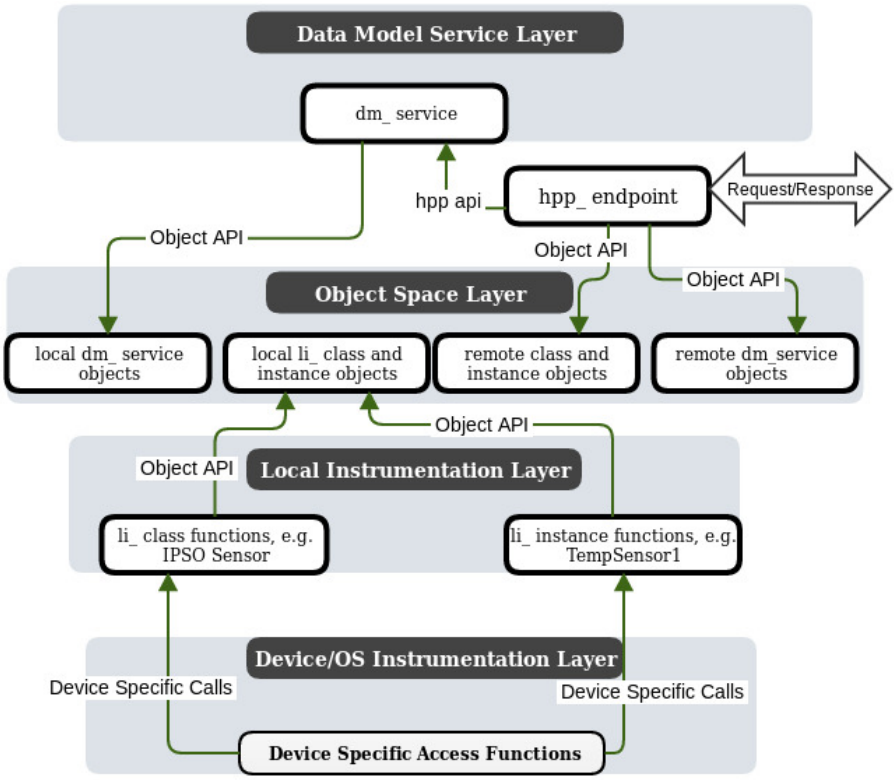

Figure 14 Node Architecture

The holistic architecture consists of the following layers shown in Figure 14 and all of which meet the requirement of being able to run on constrained sensor nodes:

- Data Model Service Layer - represents nodes and services (on node or Cloud) using defined roles based on capabilities, in order to be independent of particular node hardware and handle a range of node functional capabilities. It must also be independent of a particular data model and yet provide a simple and flexible API for a data store that allows it to hold data that can be used by different higher-level applications and services. This is achieved by the underlying object space and its simple object API.
- Object Space Layer - a data store modelled as an object space to hold its own data (from the local instrumentation layer) or data it has cached from remote nodes.

- Local Instrumentation Layer – a layer to map the hardware or OS specific functions on a sensor device to the object space layer, i.e. to represent the sensor's own data in the object space, just as for data from remote nodes.
- Device Instrumentation Layer – the device specific layer to interface with its OS or hardware.
- hpp_endpoint represents the peer service's communication endpoint, with a HPP channel per peer interaction, such as a TCP connection, to hide the specific network layer implementation details. The service code is only aware of the hpp_channel and hpp_endpoint and will not have to be modified for a new network other than for configuration settings.

These layers may not all exist in a given node, depending on its role (defined as a combination of the defined roles), based on its capabilities. A node uses the HPP protocol to exchange messages with other sensors and services. While the architectural layers are not dependent on the HPP protocol itself, that protocol is sufficiently simple for low capability devices to participate and it provides a consistent means to exchange sensor information independent of the underlying technology. The term instrumentation is used here to refer to the sensor data provided by the node (and not instrumentation for management of the node itself).

Furthermore, Figure 14 shows the data model service layer providing a higher level abstraction for node data and its use of the object space. It also shows the separation of remote peer data and local data and how this architecture, particularly the object store, allows for a cache to hold data from the local node and data from remote nodes.

The data model, object space and local instrumentation layers also treat an object's key and non-key properties/attributes separately, as many information models use keys to identify object instances (e.g. CIM) or table rows (e.g. SNMP or HBase). It is also because resource constrained devices often set keys when the class is created, which can be allocated then, whereas non-key data in an instance changes and may be read by a dynamic getter function.

4.3.1 Service Abstractions

Fundamentally, WSNs exist to provide sensor data and events (and actions for transducers, actuators) to consumers of such data as efficiently and reliably as

possible. This requires service abstractions to support the use or integration of such data (e.g. by domain experts analysing sensor data) and low level abstractions that simplify gathering and communicating the data between sensor nodes and higher level application.

The data model service layer provides roles for the type of services involved in the end to end flow of data and it exists in order to insulate the application developer from the network and node specifics. It defines nodes in terms of their fundamental role with respect to the data model and data flow, with the following defined roles (and more may be added for particular purposes, e.g. for security as discussed in 4.1.5):

- DM_SINK_SRV (receives data from peers and optionally informs its peers about data it is interested in using a *Notify* message)
- DM_SOURCE_SRV (has sensors and sends data to its peers)
- DM_FORWARDER_SRV (will pass messages to other services)
- DM_STORE_SRV (provides intermediate storage for data from remote peers, such as historic data)
- DM_AGGREGATOR_SRV (aggregates data from peer services)
- DM_MATCHER_SRV (provides results of advanced matching queries).
- DM_BOOTSTRAP_SRV (provides an identifier for the node in the DHT)

A node could play several roles according to its resources. For example, a constrained node may act only as a DM_SOURCE_SRV sending its sensor data in response to requests or unsolicited and it may not even store its own data or forward requests that it receives from other nodes. Also, a node may dynamically remove its capability as a DM_FORWARDER_SRV if low on remaining power. Note for example that the role of a DM_SOURCE_SRV aligns with the non-storing mode outlined in RPL and DM_FORWARDER_SRV aligns with storing mode.

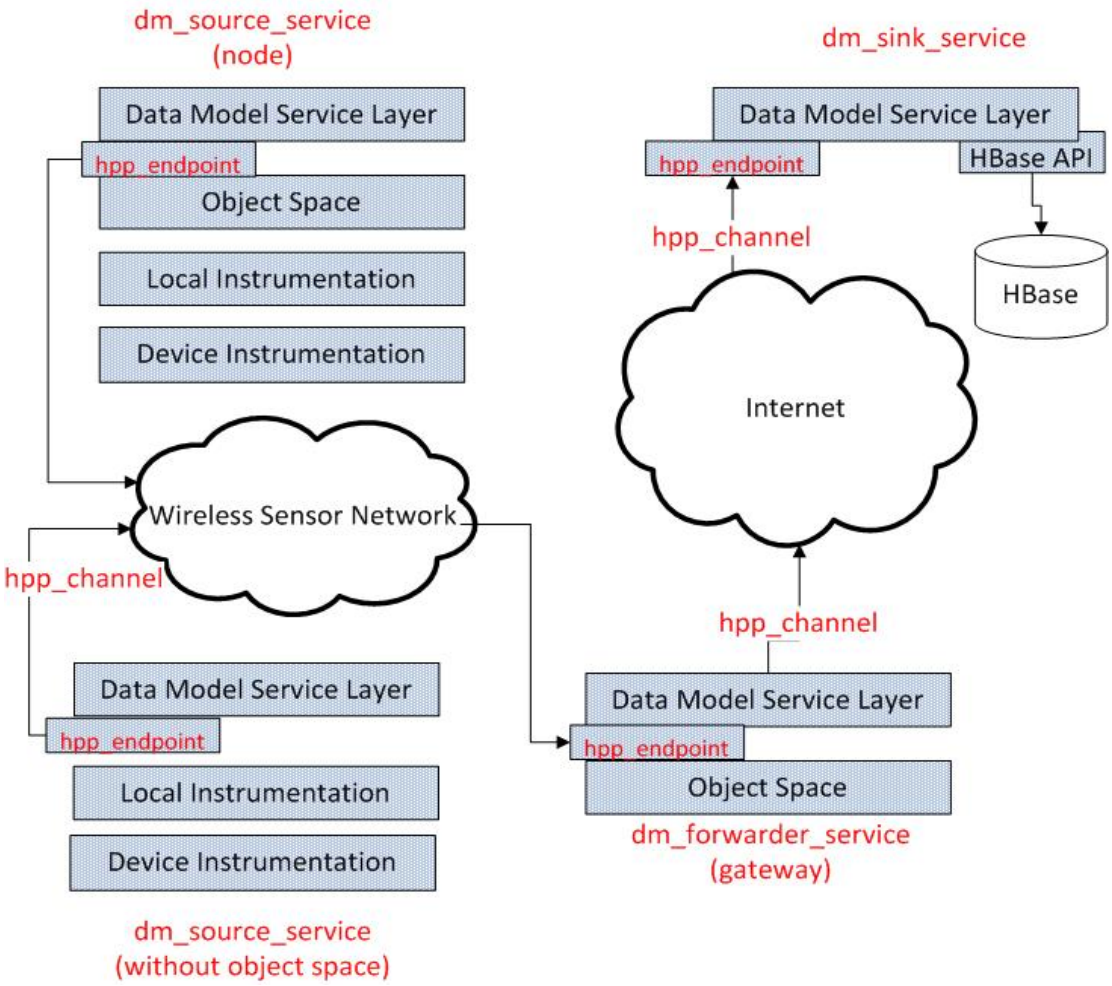

Figure 15 Holistic Architecture

Figure 15 shows the full end-to-end relationship of a number of HPP nodes, having different capabilities and roles, i.e. a DM_SOURCE_SRV without an object space that cannot hold remote data and one that can and also a DM_FORWARDER_SRV that does not provide any sensor data and so does not have a device instrumentation or local instrumentation layers. It also shows how nodes may forward data to a DM_SINK_SRV in a Cloud or remote data centres, where that DM_SINK_SRV exchanges HPP message and provides the same simple API in front of HBase that it does for the object space elsewhere. This can also allow sensor nodes and high level applications to exchange their capabilities and interests, so a node could set its sensing and communications to meet the response requirements based on the received interests, e.g. a sensor may be able to report every 15 minutes, but an application may only want an hourly report, so the sensor need only actually send a message every hour.

The data model service layer also provides a rich set of matching capabilities for a query, i.e. it can match on key attributes, non-key attributes or a wild card. Matching is implemented in the data model service layer and not the object library, as that library is transparent to the contents of the objects in it.

Note that fog and edge computing can be supported architecturally in these roles as nodes may perform actions based on events or external input or possibly aggregate the sensor data, for when aggregation and analysis may be required close to the source. For example, a sensor testing for hazardous gases must react to major events based on timely local analysis of its own and neighbouring sensor readings (current and for a past period) and also forward data to a centralised, probably Cloud, system for longer term storage and more detailed historical analysis.

4.3.2 Object Space

Inspired by the simple operations in Linda and Javaspaces and the decoupling benefits of a shared tuple space, this work developed an object space library to provide a simple object-like infrastructure suitable for implementation on resource constrained devices with object functions to support a simple shared object store and associated API. It is used to store locally instrumented data and data received from other nodes for aggregation or other purposes, as well as DHT data. The object store is non-prescriptive about how it is used to hold classes and instances, except that it requires the use of a template to hold the type of each attribute of the object and its methods. TeenyLIME [101] has shown the usefulness in WSNs of a tuple space approach. The object library is, however, not reliant on a Java runtime environment and has been implemented in C.

An object structure represents an object and is held in the object store, with a separate associated template. Each object has a lease to determine the duration the data is to be held in the store allowing for the space to remove objects if leases are not renewed, similar to concepts in Javaspaces. The need to manage memory use on constrained nodes without adding significant CPU or memory overhead for code or data and the use of a lease in the object space suggests the use of CacheL for implementing the object store, particularly on constrained nodes.

The template and instance for an object are stored separately, in order to support objects that represent a class (i.e. do not have instances and only need to be stored once, even if instances are stored from remote nodes) and to allow a range of object encodings. A node can add a template defining the information model and the names of the objects it supports to its local space and sends an add message to remote nodes (selected based on the DHT) for them to cache that data or it could just add objects immediately. For resource constrained devices it also offers an efficient way of transferring them to other nodes, where the template (or a reference) can be sent once to another node prior to the encoded object. Templates are also used to define node capabilities on a model/object basis (i.e. to specify which properties of a standard object are instrumented). The actual definition of the template is transparent to the object store, although the implementation uses a specific key value pair based definition.

The object library implementing the object space is not reliant on a Java runtime environment and has been implemented in C, with the following main API functions:

- object_add()
- object_take()
- object_get_by_handle() for direct access,
- object_get_by_name()
- object_get_instance()
- object_lease_renew()

4.3.3 Local Instrumentation (li) Layer

This layer hides the platform specific sensor implementations on a particular node and provides get()/set() functions and method prototypes for node functionality such as power off. The local instrumentation (li) layer supports local data and provides an abstraction above the device specific layers to map to the underlying node functions or data.

This layer allows developers to take advantage of efficient vendor and device specific features to access sensor data, such as reading a value from a register or an API call like get_sensor_reading(), and also allows the use of C language features such as pointers to reduce memory usage. Figure 14 also shows the

separation of remote and local data, which allows the roles identified above to be implemented, i.e. data to be transferred or stored for forwarding to another node is held directly in the object space (usually as key value pairs), whereas local data makes use of underlying hardware facilities, unless it is static and can be held in the object space.

From an information modelling perspective, it is important to note that it allows the implementation of only those attributes supported by the node (with per property structures), where the li_class_property does not make any assumption about the object it is to be put in (it could appear in more than one). This has the following benefits:

- using the attributes at the local instrumentation layer provides the flexibility required to map a rich information model to a resource constrained WSN device, by allowing for an attribute based model. This means that the data model can be built up using a local instrumentation structure per attribute to allow per attribute mapping to the underlying node functions or data. It also allows these attributes to be built into a range of higher level information model, e.g. an OMA IPSO object or SNMP MIB or CIM object.
- only the attributes applicable to a given node have to be implemented, saving code and memory, rather than having to store null entries for an object's unsupported attributes.
- the attribute can be added into tables or key value stores such as HBase.

This per attribute approach also maps particularly well to the approach taken in OMA LWM2M, which will be demonstrated in later sections.

4.4 HPP Overlay using Distributed Hash Table

A P2P overlay approach is used based on the analysis of earlier work in section 4.1.6.4. The key principles underlying the use of P2P in HPP are:

- no fixed placement of data.
- consistent handling of local and remote data with a small set of messages, aligned with the object space and easy to map to RESTFul APIs.

- all peers use the same P2P overlay network according to their capabilities, e.g. the number of identifiers (and corresponding buckets) to be stored depends on the capability (i.e. memory) of the node.
- leases are per class and instance and are set by the source, to allow nodes to cache data and to aid data consistency across nodes, i.e. they are expired based on the lease on all nodes.

Kademlia is chosen to be the basis of the DHT for this P2P overlay in the proposed holistic architecture for the reasons discussed in section 4.1.6.5. While the essentials of Kademlia node lookup and associated buckets are retained, it is used with HPP messages, rather than the specific Kademlia messages. It is also adapted to be more suitable for use in a WSN, including ideas adapted from BitTorrent, such as setting k to 8 and modified refresh behaviour. This results in the following key principles and novelty in the proposed use of a DHT:

- use of Kademlia DHT buckets for node-identifier and xor based routing, initialized with 1 bucket as in BitTorrent (not 160 as in Kademlia) to reduce the memory required.
- use of Kademlia buckets to dynamically group peers. While the main expected use of the DHT is to hold peer identifiers, where a *Get* for the key returns the closest peers to it, HPP can use the DHT to not just hold identifiers to peers, but to hold identifiers to group nodes or related data (as in a torrent). This group can be joined and retrieved using the same HPP messages that a peer uses to join or be found in an overlay. For example, in HPP, a *Get* message with a key could return a list (buckets) of items such as geolocations, wireless networks (a *Get* returns the closest networks in that list) or a list of object classes or instances such as a grouping of temperature sensors or HPP services with a given role.
- peer longevity in the cache uses a lease set by the source and HPP messages are used to reduce the overhead of co-ordination and information exchange in updating leases and buckets.

It was decided to incorporate the identifier and bucket aspects of the DHT into the Holistic Peer to Peer Protocol (HPP), rather than to port an existing BitTorrent implementation with Kademlia and use it separately. This decision was taken in order to:

- maintain a set of constrained set of messages, similar to the RESTFul architectural style in terms of verbs and message layout. For example, HPP *Get* is used to retrieve a Peer's object or its identifier and address rather than a Kademlia Find or BitTorrent GET_PEERS.
- allow the HPP commands to update the times associated with peer messages and avoid refreshing nodes that are active and so reduce the overhead of co-ordination and information exchange in updating leases and buckets.
- use the object space, lease handling and roles that are part of HPP for peer information, i.e. to retrieve and store the Peer object and delete it if its lease is not updated. The bucket is still used to hold identifiers, so its concepts of closest identifiers and identifier lookup could be used.
- use the DHT to forward to peers as part of HPP (see section 4.5.1)
- reduce the code size as the HPP commands are already implemented. Note that the BitTorrent client could not be used without modification anyway, e.g. the code associated with file sharing would have to be removed to reduce code size.

4.5 HPP Protocol Design

The Holistic Peer to Peer Protocol (HPP) is a simple message protocol suitable for resource limited nodes. It includes a small set of simple commands, as per the Linda approach, for nodes and services to add/remove instances to/from the tuple space for their capabilities, interests and data, including setting leases on the data. Applications running on top of this protocol will not require any knowledge of the underlying implementation. It has been developed to:

- support interaction between the defined service roles in the data model service layer to model the roles in the end-to-end flow of sensor data from device(s) to service(s).
- use a limited set of message types in line with the operations of the object space, such as Get, Take and with a Publish/Subscribe model, which MQTT has been shown to be useful in certain WSN use cases. The message types are also defined to make it easy to map to the RESTFul APIs used elsewhere.

- offer the scalability and resilience properties of a P2P protocol (together with leases) to handle the intermittent connectivity and mobility of nodes.
- allow an info-hash to be associated with a group of peers, separate to their individual peer identifier. A peer can request to be added to that info-hash by sending an *Add* to a peer in that group, which may be a specific node or any node according to how the group was initialised.
- support stateless relationships between services, i.e. services must renew leases for their objects with their peers.

Data added into the object space can be explicitly timestamped on addition and has a lease. This means that the data is immutable and while a node may not have stored the most recent data (and may expire it), it always has data which was valid at that time. This means it is ok to add data to lots of groups identified by an info-hash (swarms) and there is no need for a centralised store of all groups and their contents, e.g. if a node is no longer operational it will not refresh any data it added and so the lease will expire.

Two key abstractions are used on top of the network layer. These are the *hpp_channel* (the link between peers hiding the network specifics) and the *hpp_endpoint* (represents a communication endpoint consisting of channels). Using *hpp_channel* and *hpp_endpoint* provides a single API so that applications using this protocol will not require knowledge of the underlying network and means applications do not need to be re-coded for different networks. Indeed, the *hpp_endpoint* and *hpp_channel* can be used to handle non HPP traffic. This is similar to the way JXTA abstracted the underlying network.

4.5.1 HPP Forwarding and Routing

HPP can be used to forward messages to peers, but HPP can also be used with routing protocols such as RPL. When using RPL, HPP peers use the DHT to look up a peer identifier and find its IP address and port (or the closest IP address and port) and the message is then routed to that address by the routing layer. The use of a routing layer also allows interoperability for HPP Peers with nodes that do not support HPP. For example, using a border router to route between the WSN and the Internet allows a CoAP browser to read values from a CoAP node using the holistic architecture's object space (which may also support HPP to get the

same data) in the WSN. It also allows a HPP Peer with the DM_SINK_SRV role to get data from a DM_SOURCE_SRV in the WSN.

Using HPP to forward *Get* messages can take advantage of data cached from previous replies on the path to a peer, so reducing the number of hops and messages sent. It can also be used to send *Add* messages and if the number of hops is set in the message, then the reply is only from the first peer to limit the number of messages. The use of a separate routing layer does, however, mean that the HPP cache will not be used to build a reply by an intermediary peer on the path to the source peer, unless the destination IP address has cached that data using previous replies from other nodes.

Another example of when to use HPP forwarding is when the DHT stores and uses additional metadata to select the closest peer, e.g. using the time taken by the last message to that peer identifier. HPP could also be used if RPL failed to route for some reason and the message could be sent to the defined α number of closest peers from the DHT.

A further example of when to use HPP forwarding is for peers that do not have an IP address, as may occur in a WSN or for peers that do not want to be visible on the Internet beyond the bootstrap peer and only have a peer identifier obtained from a bootstrap peer. Whereas Kademlia assumes all nodes are IP connected and uses IP addresses to send messages, HPP can forward to a peer without using an IP address. In this case, that node returns 0.0.0.0 as its peer address and requests to its peer identifier are routed over IP to the closest peer identifier to its identifier. That closest peer in turn forwards to the next closest peer identifier, until eventually the message gets to the peer with a HPP channel to the node with that destination identifier, which is probably the bootstrap it connected to.

4.5.2 HPP Actors

HPP considers 3 actors:

1. **Service** (hpp_service) – runs on a **node** and is defined as its set of roles (sink, source, forwarder, store, aggregator, matcher, bootstrap) from the data model service layer.

2. **Peer** (hpp_peer)– runs on a **node** and is defined as a set of capabilities to handle HPP messages. These capabilities are advertised to other **peers** in the *Hello* message
3. **Node** - represents the WSN node. It contains one or more **peers** and has one or more service roles. It also has an object holding node information, e.g. firmware version, with an object id, which can be retrieved over HPP.

A service's main function is to hold information on services and data from services on other nodes. It is modelled as an object that has the following attributes:

- DM_ROLES, defined as per DM_SERVICE
- DM_SERVICE_NAME
- DM_HANDLE (object id)
- DM_VERSION
- DM_LEASE

A hpp_peer holds its capabilities in terms of HPP messages it can handle, which is a lower level view than the roles of a service. Note that a hpp_service with a DM_SOURCE_SRV role has to have the capability to accept HPP_GET to retrieve data from a hpp_peer, but not necessarily a HPP_ADD to accept data from a hpp_peer. The hpp_peer does not know of the roles in use for hpp_services on that node - that is the concern of the hpp_services running on the node. The hpp_peer exists primarily for the connectivity between nodes and it is assigned an identifier in the P2P overlay network – as such its relationship to a service and node is similar to that of the Kademlia client running for a BitTorrent client on a laptop or server node.

A hpp_peer exchanges HPP messages with another hpp_peer, with the following rules:

- Every peer must support the HPP *Hello* message and respond with its identifier (if known) and its capabilities, i.e. the HPP messages it handles. Hello is deliberately simple to run on very limited nodes.
- Every peer should handle at least a HPP *Get* message for its peer instance, returning up to 8 closest node-ids (or only its own node id if it is only a DM_SINK_SRV).

- Peers may support any of the other HPP messages, which are the capabilities in its *Hello* reply.
- A new peer joins a HPP overlay network by sending a *Hello* message to a known peer.
- HPP *Get* requests for keys may be passed from node to node in the absence of other routing to a node. If a node has the requested data, then it sends that back to the requester, otherwise it forwards the request to the node with the "closest" identifier in its routing table.
- A peer may choose to not accept connections for security reasons, e.g. a source or sink may only make outbound connections.
- A peer will *Get* its closest peers and send a *Get* to peers of interest to discover the classes and instances on that peer, avoiding the need for a centralised Resource Directory as in CoAP.
- A peer uses the peer object's lease (and the cache which implements it) to replace the republishing period in Kademlia.
- Peers can handle a *Get* specifying the following type of match:
    - EXACT to specified attributes and values, e.g. keys for an instance
    - ALL_HANDLES
    - ALL_CLASSES
    - ALL_INSTANCES (for Discovery)
    - WILDCARD

    The *Get* Response can contain object identifiers (handles) or full classes and instances

- When processing a *Get* reply, the object data in the reply can be cached in exactly the same manner as in processing an *Add* message, if that Service's role allows it to cache data received remotely. This is another example of the design allowing code re-use in order to reduce code size.

Note that as a Peer is an object in HPP, its lease can be renewed (and it will remain in the routing table for that lease period) by using an *Add* message or a specific *Get* for it as per Kademlia. The use of a lease request/grant means the two peers can operate more flexibly than the fixed refresh times in Kademlia.

Resource Discovery is described in section 4.5.8, which shows the use of a *Get* for all objects on a known peer. This is analogous to the well-known URL in CoAP on a node, but there is no need for a centralised Resource Directory(RD) as in CoAP, due to the use of a DHT to find peers and the HPP *Get* message to retrieve the objects/resources.

Techniques to take advantage of the knowledge of peers in this overlay network and map efficiently to underlying routing functionality such as RPL or WSN specific networking will be possible once this overlay is in place, e.g. Kademlia peers can route messages through low-latency paths.

4.5.3 HPP Message Header

HPP Request Messages consist of command, message header and payload object. Command is one of the allowed commands *Hello*, *Bye, Get, Add, Take, Notify*.

Command	Message Header	Object
Get	msgId=2 senderId=AAA name=peers/peer Lease=60	peerId=ZZZ

HPP Replies are designed to be the same as the original request, with the addition of a status field. This allows shared message handling code to reduce memory use, e.g. caching the data in a *Get* reply uses the same code as an *Add* request.

Command	Message Header	Object
reply=Get status=Ok	msgId=2 senderId=BBB name =peers/peer objectHandle=1001	closePeers =YYY, TTT closeAddresses=a.b.c.d:7014, e.f.g.h:7014

The protocol is designed to be encoded as a set of strings for ease of use on higher level devices, but it is also designed to be binary encoded for efficiency. In both cases, messages will consist of blocks. In the string encoding, the blocks use specific delimiters, e.g. “header”: or “object”. In the binary encoding, every block has the same preamble of a block type, a block length and a block id. This allows the device to ignore blocks it is not interested in.

The first block is always the header block and it is always preceded by the HPP command (or message type). A hpp_peer parses the command to determine if it should process this message.

The header must have:

- msgId - unique to originatorId and not changed when forwarded
- senderId or replierId (peerId)

The header may have:

- version- HPP version and must be returned in the *Hello* reply
- caps - HPP peer capabilities in terms of commands it supports and it must be in the *Hello* reply
- type - encoding of message content
- name - class name, intended for use in class template
- group - object group
- handle - set by a resource/object creator, e.g. peer for objects it creates - it is unique on a peer or is a UUID. Its uniqueness for a local peer relies on its use with a peerId which is globally unique
- lease - object lease requested (or granted in a reply)
- originatorId - only included if not the same as senderId
- destinationId - only included if using HPP forwarding (see section 4.5.5.4)
- hops - the number of message hops allowed before a message is no longer forwarded (this is decremented at each hop)

originatorId and msgId never change. The msgId is included in the reply and if forwarded. The originatorId is included if forwarded and is how HPP can route the reply to that peer from the peer with the destinationId.

In addition to the command and header, a message can consist of the following types of blocks:

- **Object**. This holds object (class/template or instance) content between SRV_OBJECT_BEGIN and SRV_OBJECT_END delimiters. The content consists of key value pairs of “attr=attribute” name for a template/class

and “attribute name=value” for an instance. Attribute names used in the instance must match those in template.

Credential. This is transparent to HPP itself and driven by the requirements of the peers. The nature of the credential required by a peer to communicate with it will be returned in the *Hello* message or the *Hello* may require a particular Credential block and replies with an error otherwise. In other messages than *Hello*, the specific credential block contents will be carried. This in line with the approach outlined in 4.1.5

Like Kademlia, every message includes the sending node’s identifier. In some cases, the originatorId is included in the message, i.e. the node with the sensor(s) whose readings are being carried. This additional overhead is considered more acceptable at this higher layer than would be appropriate in a WSN routing layer.

For comparison, the CoAP header uses 4 bytes for its encoded header; 2 bits for version, 2 bits for message type req/ack, 4 bits for token length, 2 bytes for message code, 2 bytes for its message-id and 0-8 bytes for a token followed by a variable number of bytes for Options.

For the case that the 20 bytes of a Kademlia identifier is too expensive, a peer specifies the bytes it can use for an identifier in its *Hello* message to its bootstrap peer, e.g. a gateway on the WSN. That bootstrap peer will respond with an identifier of that number of bytes and will include its own full identifier as the reply identifier to the *Hello* (and only to the *Hello*). All other messages from the limited peer will use the number of bytes in its own identifier and will receive replies with that number of bytes in an identifier and it will have a routing table that is a sub-tree of the full 160 bit space. This shortened identifier is valid within the range of the bootstrap peer that issued it (and that bootstrap peer will expand it for any communication outside that WSN to the rest of the overlay in the IoT).

4.5.4 HPP Message Types (or Commands)

HPP has the following limited set of message commands:

- *Hello*. This message is used to join a peer overlay, by sending a peer’s capabilities and its peerId (if it has one) to a well-known peer in that

overlay. A peerId will be returned if one was not included in the request and the receiver is a bootstrap peer.

- *Bye*. This message just contains a message header and the receiver removes any peer information explicitly ahead of lease expiry.
- *Get*. This message retrieves an object (class or instance) by specifying an object handle or a match type with range of attributes as specified above, including wildcard matches for a given attribute. A service can send a *Get* with an object name and template/instance to be used to find the object for other services or objects representing data on a node. If a peer can send directly to another peer, then it can retrieve an object using the object handle in that node's object space, by simply specifying "handle"="someValue" in the *Get* message header. This makes *Get* similar to Kademlia's Find message when getting services and the Find_Value for other data. A *Get* can also specify an info-hash or group identifier if the object was added to that group identifier. On getting a reply to a Get "Peer" message, a peer must check the peer ids as in a Kademlia "find round".
- *Take.* The *Take* message can use the object handle or a match type using the attributes of the object to remove. It is not simply a Delete, as the reply returns the object, so it can be added back (with a new timestamp) and may be used to handle concurrent updates, i.e. once an object has been taken from an object space, it is not there to be changed by another peer. The *Add* and *Take* messages carry the originatorId if it is not the same as the senderId in the message as only the originator can change the object if it has been distributed over many nodes and if the originator does not renew its lease, then it will expire according to its lease.
- *Add*. This message is sent to a peer to add new classes, new instances, for updating an object with new values and to call a method in an object (adds a method with the parameters to use). It contains the object to create or update. For an instance object, it will be used to build an instance of the object in the object space. If it contains a template object, it will be treated as a class that can be referenced by other adds, e.g. to avoid having to include the full template for future adds. Its object handle in this peer's object space will be set in the reply.

The *Add* message may contain a lease. In general, an object or its lease is updated by a new *Add* containing its object handle and new contents. Optionally a *Take* could have been sent to remove the old instance before the new *Add*.
An *Add* message can also specify the identifier of a group to add itself to, e.g. a group of nodes in a certain area or a group with a certain sensor type – this is analogous to a BitTorrent client adding itself into a swarm with an info-hash of a torrent.

- *Notify*. This message is designed for the alert functionality of devices, similar to observe in CoAP. It tells a peer that this peer is interested in updates to an object for a lease period. A *Notify* message tells a peer of an interest in an object and it will send that peer the add/take message for a matching object when it is added/removed (this removes the need for a separate report/event message). To implement *Notify* functionality on connectionless systems, the peer is polled with a *Get* or will piggyback *Notify* information in the next response to a message from the peer that sent the *Notify*. A message generated as a result of an earlier *Notify* will simply contain the relevant message be it *Add*, *Take* as an object

As above, all peers must support *Hello*, but peers may support only *Get* or *Add*. *Hello* provides the ability to join a P2P overlay and exchange peerIds and HPP capabilities. The *Get* and *Add* command types map well to the GET, POST, PUT, DELETE commands found in HTTP APIs and CoAP using the RESTFul approach. A *Notify* primitive has been added for the alert functionality of sensor devices, similarly to how CoAP added observe. There is no specific Action message type, as this is specific to an object and would be a method defined in that object. It would be invoked by an *Add* with the arguments specified for the object.

Hello/Bye will be handled by any Peer. *Get* messages can be sent to any peer to retrieve closest PeerIds, similar to a Kademlia Find, although a service with only the DM_SINK_SRV role will only return its node identifier (as it has no other data to *Get*). *Add, Take* and *Notify* messages will only be handled by services that have Forward, Aggregate or Store roles and a service will reply with an error if it cannot handle the message.

4.5.5 HPP Message Flows

This section considers the message flows between HPP peers, including the use of HPP to forward messages. This flow is considered as point to point between peers in this section. Peers may, however, have knowledge of several closest peers and a peer could send messages to all of its peers at the same time in a manner like multicast, e.g. to add its data instances to multiple peers, or it could actually avail of multicast facilities in the lower layers of the stack to do this.

4.5.5.1 Initialisation - Hello Exchange

A node wanting to join a HPP network must get a PeerId. If it does not have a PeerId on starting up, it will send a *Hello* to at least one known bootstrap peer that is able to allocate a PeerId for it. The *Hello* message contains its encoded capabilities and may contain its PeerId. If the *Hello* does not have a senderId, a receiver with the bootstrap role will check any message credentials provided and if they meet the bootstrap's requirements, it will process the *Hello* request. If the *Hello* request contains a PeerId, the receiving node adds this to its DHT and it replies with a message header containing the version and capabilities it supports. If the *Hello* did not contain a PeerId, then this peer returns a message header and also a peerId to the sender if this node can issue an identifier based on its capability (or authorisation) - if it returns an id, then it supports DHT. The PeerId issued will be returned in an object block for a peer object. If this peer sent no other message within a defined period, e.g. 1 hour, then it will send a *Hello* to k=8 nearest nodes, similar to the Kademlia use of Ping to refresh its routing tables.

This peer then adds (or takes) instances of its objects, with a lease per object, its capabilities and its interests in other objects with the object space layer.

4.5.5.2 Retrieving Closest Peer Information

Once the *Hello* reply has been received, a node can send a *Get* for the peer's object (identifier shown as zzz) and its k neighbours will be in the reply, as below:

Command	Message Header	Object
Get	msgId=2 senderId=AAA name=peers/peer Lease=60	peerId=ZZZ

Command	Message Header	Object
reply=Get status=Ok	msgId=2 senderId=BBB name =peers/peer objectHandle=1001	peerId = ZZZ, closePeers =YYY, TTT closeAddresses=a.b.c.d:7014, e.f.g.h:7014

4.5.5.3 Data Transfer – Adding/Retrieving Classes and Instances

Once a peer has exchanged a *Hello* with a Peer and has a PeerId, it will add the following instances to that peer.

- Peer instance with its address information
- Node instance representing the node, e.g. its serial number, firmware version
- Service instance representing its data model service roles

A service can send an *Add* message to peers directly or to a specific info_hash for a group (swarm) of peers with an object, e.g. it could add itself to a group of peers in a geographic region. When an object is looked up, for a *Get* or *Add*, the request is routed by PeerId or Info-Hash

It is assumed a Class object (or template) is unique if no object specification string is provided and the handle is then used to identify it on that peer. If a Class object is common, e.g. from a defined object model, then the object specification string, i.e. namespace, should be checked when it is added to see if it was already added and if so, the handle for it is returned. It is expected that nodes and services will add their own namespaces for class definitions they want to share, e.g. IPSO or a pre-assigned UUID could be passed in the Add message which is shared by data model service implementing that Class.

For example, Figure 16 shows how, after it has sent its Hello message, a node operating as a DM_SOURCE_SRV service adds its service and node classes (or templates) and instances to a DM_STORE_SRV service on a node able to cache data.

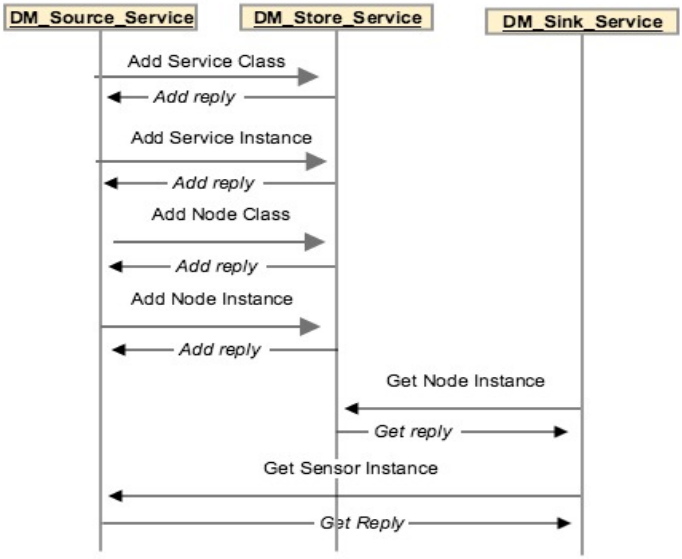

Figure 16 Sample HPP Service Interaction

It also shows how a DM_SINK_SRV service queries this DM_STORE_SRV for its capabilities and then its node data, which may be returned from the DM_SOURCE_SRV or an intermediate DM_STORE_SRV (if cached there). Other interactions are possible, e.g. the DM_SOURCE_SRV service could also add its sensor class and instance to the DM_STORE_SRV service, so the DM_SINK_SRV can retrieve the cached data and so reduce the transmission cost.

4.5.5.4 HPP Forwarding

HPP forwarding is used when the destinationId is set in the message and the peer has the DM_FORWARD_SRV role. In this case, a peer receiving a message will send to that destinationId peer if it cannot service the request itself (e.g. it has not cached the requested object from handling a previous *Get*). If it has a HPP channel to that destinationId or a bucket entry with the IP address for it, it will use these to forward the request and if it does not then it will forward to the “good” peer in the DHT with the closest identifier to the destinationId. If the hops parameter is set in the request, this will also be used to determine whether to forward, i.e. it will only forward if hops is greater than 0, and hops will be decremented in the message if forwarded.

4.5.5.5 Lease Renewal

A lease would be renewed by sending a new *Add* containing the object handle for the lease to renew. As well as renewing leases on data objects added to a peer such as for sensor data, lease renewal is also required for DHT based peer objects. The peer object's lease replaces the republishing period in Kademlia (and not the bucket refresh period). A peer requests a lease and the bootstrap grants it per its policy. If the lease is not renewed and no message is seen within the lease, then the bootstrap tries to refresh the lease by sending a *Get* to the peer and removes the peer object if it does not reply.

4.5.6 Decentralised Control

A key part of the holistic architecture is to use P2P to support decentralisation and give control to the peers themselves, with support for co-ordination. This is done in order to enable scalability (including for management) and robustness. This can be shown in the cases of security and lease management, although this may still require the use of policies which must be distributed or built into each peer.

In terms of security, the bootstrap peer (or peer contacted to join a group) will use its own security criteria/policy to process the *Hello* and determine whether the node is allowed to join the overlay or group, per the initial exchange in 4.5.5.1. This may include setting up a separate exchange of security information (known to both nodes) or a check on a credential or token provided. The fact that the peers know of other peers in the overlay (or group) allows for the nodes to share this authentication of new nodes, e.g. it may use mechanisms such as those provided in JXTA, where nodes "vote" to allow a new node to join. It is also worth noting that capabilities can be defined that nodes can share in the *Hello* message and then use to negotiate the security mechanisms to use (similar to the use of HTTP accept headers in negotiating the format of content to exchange). Another aspect of the initial exchange is that unlike CoAP, HPP accommodates nodes that refuse to accept incoming connections. For example, if it is a DM_SOURCE_SRV only, it can connect out to a peer, exchange its *Hello* message, add its objects and then allow requests from the peers it connected to.

In terms of lease management, both the requesting and granting peer control the use of the lease. The peer granting the lease and storing the object can decide on the lease it grants based on its own policy or available memory. The requesting peer has control over which peers it adds its object to, but also in the event of not being granted its requested lease as it can then renew the lease granted more frequently or it can add its object(s) to other peers until it gets the lease it wants.

4.5.7 Energy Consumption

HPP is an application layer protocol that is designed to be used in WSN and IoT scenarios. Energy conservation is important in WSNs as discussed earlier and HPP will benefit from the energy conservation mechanisms implemented in lower layers of low power wireless stacks. As with other application level protocols like CoAP, the energy consumption of the higher layers will depend on the implementation of the application itself. The following aspects related to energy conservation are included in the design of HPP:

- Low HPP overhead. The only additional HPP messages beyond normal traffic are *Hello* and *Add* to renew leases. The message overhead in maintaining routing tables is kept low by the use of normal traffic, such as *Get* requests, to update the information about other nodes held on a node and by the ability to tune the refresh periods in the absence of traffic, e.g. according to the expected failure rate of nodes.
- Caching. The use of caching (if using HPP to forward as per 4.5.1) allows nodes to respond if they have cached the requested data, avoiding the need to forward the request to the originating source.
- Binary encoding. HPP is designed to consist of blocks of data that can be encoded in string or binary format, with the binary format to be used to reduce the size of messages.
- Identifier size. The 160 bit identifier size can be reduced in the scenario of a small WSN, where the nodes only communicate beyond the WSN using a bootstrap node. Within the WSN, the identifier only needs to be the size required to uniquely identify nodes, while the prefix of the bootstrap node is combined with the reduced node identifier for use beyond the WSN.

4.5.8 Resource Discovery

If a peer does not know where a peer (or object) is, it will select a bootstrap node (to which it sent its first *Hello* to get its PeerId) or the closest peer(s) it has in its routing table. It will then send *Get* for that PeerId (or object) to the selected peer. For example a *Get* is sent from peer with identifier 123 to a known peer with identifier 999 (this identifier will be 20 bytes in a real example):

```
{
        "get": {
                "header": {
                        "msgId": "456",
                        "senderId": "123"
                },
                "object”: {
                        "type":"instance",
                         "class":"peer",
                        "peerId": "789"
                }
        }
}
```

This will return the peer’s address information, as for example

```
{
        “reply”: “get”
        “status”:”ok”
        "header": {
                "msgId": "456",
                "senderId": "999" // The responder’s id
        },
        "object”: {
                “type":"instance",
                 “class":"peer",
                “key”: { “peerId”=”789”},
                “close_peers”: ”999”,
                “close_addrs”=”127.0.0.1:1111”,
                “address:”127.0.0.1”,
                “port”=”2222”
                “version”:”1”,
                “caps”:”63”,
                “lease”:”0”
        }
}
```

Hence, discovery of objects on a node can be performed by doing a *Get* for its peerId as above and then a *Get* can be sent to that peer's address with handle=ALL in the header.

4.6 Summary

This chapter has presented an analysis of previous research work and used that to propose a set of requirements and a holistic architecture to meet those requirements. The architecture is inspired by concepts from tuple spaces, overlay P2P networks and caching. It also showed the alignment of HPP with the architectural requirements and layers, particularly the use of leases in the cache. The analysis in this chapter also presented and explained the architectural and design choices which have resulted in the following contributions:

- a set of abstractions and layers of the holistic architecture and each of their roles in the architecture, covering the varied roles of the entities in an IoT system from constrained devices in a WSN to Cloud services.
- tuple space concepts being used to implement simple libraries on constrained devices to support the data model (DM) service layer and the local instrumentation (li) layer to support remote device data and local device data.
- the design of a novel cache algorithm using leases in its replacement policy, called CacheL suitable for constrained nodes and which was inspired by the Clock algorithm.
- the design of a simple application layer protocol, termed the Holistic Peer to Peer (HPP) protocol using a simple set of commands inspired by Linda and supported by a Distributed Hash Table (DHT) to identify nodes and groups across the end-to-end flow in IoT.
- the use of a DHT derived from Kademlia and its use of HPP messages and object store.

The next chapter will present protype implementations of this holistic architecture for Linux and Contiki3.0 platforms. These implementations were performed in order to then evaluate the components in the holistic architecture.

5 Implementation

The methodology used to determine the validity of the research work and the resulting holistic architecture was to develop a prototype implementation for Linux and then port this to Contiki3.0, sharing as much code as possible. The Linux implementation was developed first to make debugging and scalability testing easier, but it also represented more capable systems for the architecture to be used on. The Contiki3.0 implementation was used to show the implementation was possible on a constrained node in a WSN, meaning it is likely to be able to run on or integrate with nearly all systems. This is also important to demonstrate the feasibility of implementing a P2P protocol and DHT on a constrained WSN device, as a number of sources propose specific separation between WSNs and a P2P overlay, such as in section 3.8.4, due to the expected difficulties in such an implementation.

In particular, it was important to provide a quantitative analysis of the size of the code and data required in order to meet the requirement that the resources on a constrained node could support this architectural approach. It was also important to provide an implementation that could be used to produce assessments, albeit qualitative, of whether the abstractions made it easier to develop software for constrained node devices and to assess the portability of that software. Finally, implementations allowed quantitative evaluation of components in the architecture, e.g. the hit ratio of the CacheL algorithm and how the time to process DHT lookups changes as the number of nodes changes.

The implementations in this section were carried out on Linux and on Contiki3.0 and included the holistic architecture layers comprising the data model service layer, local instrumentation layer and object space (using the CacheL algorithm), with supporting libraries for memory utilities, doubly linked list, hash and leases. The DM_SINK_SRV, DM_SOURCE_SRV, DM_FORWARDER_SRV, DM_STORE_SRV and DM_BOOTSTRAP_SRV roles in the data model service

layer were implemented, but the DM_AGGREGATOR_SRV and DM_MATCHER_SRV roles were not. The implementations also included libraries for building and parsing the HPP messages in string form. The implementation did not include a full implementation of the use of a shortened identifier tied to the bootstrap peer (as per section 4.5.3) and the implemented bootstrap returned a randomly generated identifier. A peer did retry the *Hello* message to a known bootstrap peer for a set number of attempts, but the implementation did not then select another bootstrap to try, as HPP allows. It also included a partial implementation of the *Notify* message and use of the info-hash sufficient to validate the functionality. The nature of what was implemented for the different tests regarding the information model, DHT, HPP and the test sequences will be explained in those sections.

Specific implementations were performed for:

- The DMTF CIM and OMA LWM2M data models on the Contiki3.0 OS to demonstrate the flexibility of the holistic architecture layers. These implementations integrated with the erbium and CoAP implementations (er-rest-example) on Contiki [49], which aims to be memory efficient and provide convenient APIs. This implementation was itself utilised in the LWM2M and IPSO implementation in Contiki3.0 [148]. This implementation of the OMA LWM2M was also modified to use the holistic architecture's data model service layer and object space. These implementations allowed both CoAP and HPP to access the objects implemented as in Figure 17.
- A DM_SINK_SRV service. This was written in Java and integrated with HBase to demonstrate that the abstractions work in the end-to-end manner envisaged.
- The CacheL algorithm in Java and C. The C implementation was included in the object space library on Contiki. The Java implementation allowed the CacheL algorithm to be compared to a Java LRU implementation using the Yahoo Cloud Server Benchmark (YCSB) [149].
- HPP including a DHT in C that ran on both Linux and Contiki3.0. The DHT implementation used Kademlia approaches to create and compare DHT identifiers, for the selection of closest nodes and to create and manage bucket functions, e.g. to place closest peer identifiers in the

correct k-bucket. The information for peers in a bucket was held in a "Peer" object in the object space store, with a lease, like any other object.

5.1 Linux and Contiki Implementations

The code for the local instrumentation (li) layer, data model and object space, supporting libraries (memory utilities, doubly linked list, hash, lease), the DHT and the message building parts of the HPP protocol was implemented initially in C on Linux. This approach allowed the abstractions and design to be refined and the code to be debugged and tested more easily, using standard tools such as gdb and valgrind. It also allowed easily starting a number of Linux based peers, which would communicate to test the implementation of the HPP messages, DHT and the supporting data model and object space layers.

These Linux implementations were then ported to Contiki. In order to test the IPSO and OMA LWM2M models. The implementation included part of the pre-existing Erbium-REST implementation example [10] to handle the CoAP message parsing, but the objects were stored using the data model service layer and underlying object space as per Figure 14. This approach allowed these items to be tested on hardware with a supporting REST infrastructure and for the port to use existing Contiki libraries, i.e. objects stored in the object space could be accessed using CoAP via the data model service layer. The code was run in Contiki's Cooja simulation environment as a WiSMote [150], using an MSP-430 processor with 128KB of Flash Memory and 6KB of SRAM. A reduced version with just CoAP and the holistic architecture's layers (without OMA LWM2M) was also run on a Sky WSN mote using an MSP-430 Microprocessor with 10K RAM and 48K Flash. CoAP "resources" were created which integrated the cache into the holistic architecture, e.g. a DM_SOURCE_SRV was created and key value pair objects were sent to a DM_STORE_SRV.

The same codebase was able to run on the constrained nodes and more capable Linux nodes, as per the design goal for the architecture. This approach of running in two environments also showed that the architecture and its abstractions worked across Linux and constrained nodes.

5.2 HPP Implementation

5.2.1 Channel and Endpoint Communication Layers

The implementation used a *hpp_endpoint* abstraction to represent the peer service's communication endpoint, with a *hpp_channel* abstraction per peer interaction (such as a TCP connection) to hide the specific network layer details. The value of the *hpp_endpoint* and *hpp_channel* abstractions can be seen in the simplicity of the code below, with the endpoint handling the channel initialization, socket listen and message fragmentation. Other functions use *hpp_endpoint* and *hpp_channel* for message exchange and to update peer bucket statistics for each message. The following outline code is all that is required for a peer service to start receiving messages from other services:

```
rv = hpp_endpoint_check(endpoint_ptr);
if (rv == 0) {
    channel_ptr =
    hpp_endpoint_accept(endpoint_ptr);
} else if (rv > 0 ) {
    hpp_endpoint_get_messages(endpoint_ptr);
} // timed out with no data, so loop again
```

The following code shows the socket handler which had to be written specifically for Contiki using its event based model, as it did not support the select operations used in the Linux implementation. A separate call-back handler was also written in the socket handler to process received messages. This call-back simply passes the channel (including a pointer to the message buffer) and endpoints to a method to process messages (see below for example of *Hello* message processing).

```
// Handles tcp socket events - received input
// handled in a callback
static void
socket_event_handler(struct tcp_socket *socket_ptr,
         void *ptr,
         tcp_socket_event_t event) {
  hpp_channel_t *channel_ptr = ptr;
  if(event == TCP_SOCKET_CONNECTED) {
    int fd = 1;
    if (socket_ptr->listen_port != 0) {
     channel_ptr->dest_addr = NEW_V_DESC(char,
                           IPV6_MAX_ADDRESS_LEN+1);
     hpp_socket_listen_for_connection(channel_ptr);
```

```
     if (channel_ptr->dest_addr != NULL) {
              ip6addr_to_str(
                          &socket_ptr->c->ripaddr,
                          channel_ptr->dest_addr,
                          IPV6_MAX_ADDRESS_LEN+1);
              hpp_channel_t *new_ch_ptr =
                   hpp_endpoint_accept(
                       channel_ptr->endpoint_ptr);
         }
    }
    hpp_channel_connected(fd, channel_ptr);
  } else if(event == TCP_SOCKET_DATA_SENT) {
    int left_to_send = channel_ptr->send_size -
                       channel_ptr->send_index;
    if (left_to_send > 0) {
        if(socket_ptr->output_data_len == 0 &&
           socket_ptr->output_data_send_nxt == 0) {
            if(left_to_send >
               socket_ptr->output_data_maxlen) {
                 left_to_send =
                   socket_ptr->output_data_maxlen;
            }
            if(left_to_send > 0) {
                send_buffer(socket_ptr,
                            channel_ptr,
                            left_to_send);
            } else {
                channel_ptr->send_index = 0;
                channel_ptr->send_size = 0;
            }
        }
  } else if(event == TCP_SOCKET_CLOSED ||
            event == TCP_SOCKET_TIMEDOUT
            event == TCP_SOCKET_ABORTED) {
        hpp_channel_deinit(channel_ptr);
    }
  }
}
```

Even though this code only needed to be written once, it still shows the value of the channel abstraction. Also, this code frees the application developer from some Contiki specific event handling, such as registering another socket with a new channel pointer when the current socket connects in order to allow another connect to be accepted. Another Contiki specific aspect was sending one buffer in

multiple tcp socket buffers if the message buffer exceeds the size of the tcp socket buffer.

Furthermore, this code shows the use of the NEW_V_DESC macro which is used to allocate memory – on Contiki this uses fixed buffers, whereas on Linux it uses the malloc() library call.

5.2.2 HPP Message Processing

HPP messages are processed according to their type in a call-back called from the socket handler with the channel and endpoint information. As an example, the following code shows how a *Hello* message is processed. As per the description of *Hello* in section 4.5.5, the following code for process_hello() shows the check as to whether the peer has a bootstrap role, so that it can allocate an identifier for the peer that sent the request. In the case where no identifier is found in the *Hello* message by a bootstrap peer, then it creates a new peer object with that peer's information in the object space. The new or existing peer is then also added to, or updated in, the DHT Kademlia bucket for it using hpp_add_peer().

The call to hpp_new_peer_id() passes in an id_metric parameter, which can specify a metric to use in allocating an identifier, e.g. to use the time it was sent to determine how close the identifier will be to that of the bootstrap node. Another option is whether to return the full 20 byte DHT identifier or only a subset of the identifier, assuming it will only be used in a local WSN unless routed through the bootstrap node, where the full identifier is built when forwarded. The bootstrap is responsible for allocating new peer ids and it may do so randomly or do so from a range with a pre-set prefix for this bootstrap.

Similar to the use of the NEW_V_DESC macro above in the socket code, note the use of hpp_str_cpy() which is used to copy a string into a new location – on Contiki this is compiled to use fixed buffers, whereas on Linux it uses the malloc() library call.

```
int process_hello(hpp_peer_info_t *local_peer_info_ptr,
             hpp_peer_t *remote_peer_ptr,
             hpp_msg_header_t *msg_hdr_ptr,
             hpp_channel_t *channel_ptr,
             struct timeval *now_ptr) {

     int version_to_use;
     if (local_peer_info_ptr == NULL) {
            return (-1);
     }
     version_to_use = msg_hdr_ptr->hpp_version;
     if (msg_hdr_ptr->hpp_version >
                        local_peer_info_ptr->version) {
          version_to_use =
                        local_peer_info_ptr->version;
        }
     if (STRCHKNULL(msg_hdr_ptr->msg_sender_id)) {
            if (local_peer_info_ptr->bootstrap ==
                                NON_BOOTSTRAP_PEER) {
                 return (version_to_use);
            }
            int rv = hpp_new_peer_id(
                        local_peer_info_ptr->id,
                        remote_peer_ptr->id,
                        DHT_ID_NO_METRIC);
                if (rv == -1) {
                   return (-1);
                }
                hpp_str_cpy(
                        &msg_hdr_ptr->msg_sender_id[0],
                        local_peer_info_ptr->id,
                        DHT_ID_LEN+1);
        }         }
        remote_peer_ptr->listen_port =
                             msg_hdr_ptr->listen_port;
        hpp_add_peer(remote_peer_ptr->id,
                  local_peer_info_ptr->id,
                  remote_peer_ptr,
                  HELLO_COMPLETED,
                  now_ptr);
        return (version_to_use);
}
```

5.2.3 DHT Implementation

The main interaction with the DHT is to process *Hello* requests to allocate identifiers and to update peer information on receiving messages. As per the description in section 4.5, a peer only needs to know the address of a peer in the P2P network to which it can send a *Hello* message to join that network (subject to having the required security credentials). The *Hello* processing code above calls hpp_add_peer() to add the peer with its new identifier (if this peer has a bootstrap role).

The code extract below shows hpp_add_peer() which is called to add the peer to the DHT. Firstly, it checks if this id/peer is already in a bucket and if it is, it checks if it was received on this endpoint. If it is not in the bucket, then it calls add_peer_to_bucket(), which will add to a bucket with sufficient space left or replace a bad peer in the bucket or split the bucket, as described for Kademlia earlier. The function add_peer_to_bucket() uses the object space as a cache for the peer object if there is no room in the bucket.

Once the peer has been added to a bucket, it can start receiving messages from other peers and it will update the peer's information (such as the time for message received). Note that as a peer is an object in HPP, its lease can be renewed (and it will remain in the routing table for that lease period) by using an *Add* message or a specific *Get* for it as per Kademlia. The use of a lease request/grant means the two peers can operate more flexibly than the fixed refresh times in Kademlia. A *Get* for a peer object will return the object from the object space, including the 8 closest peers.

```
int hpp_add_peer(const unsigned char *new_id,
         unsigned char *my_id,
         hpp_peer_t *peer_ptr,
         char hello_completed,
         struct timeval *now_ptr) {
    struct bucket *bucket_ptr;

    peer_ptr->hello_completed = hello_completed;
    bucket_ptr = hpp_get_id_bucket(new_id, my_id);
    if(bucket_ptr == NULL) {
        return (-1);
    }
    hpp_peer_t *existing_peer_ptr =
                find_peer_in_bucket(new_id, bucket_ptr);

    if (existing_peer_ptr != NULL) {
        if (peer_ptr == existing_peer_ptr) {
            return 1;
        } else {
            if (peer_ptr->hello_completed ==
                                    HELLO_NOT_COMPLETED) {
                return (1);
            } else {
                return (-1);
            }
        }
    }
    bucket_ptr = add_peer_to_bucket(new_id,
                                    bucket_ptr,
                                    my_id, peer_ptr,
                                    now_ptr);
    if(bucket_ptr == NULL) {
        printf("add_peer-no bckt\n");
        return (-1);
    }
    return (0);
}
```

5.3 Data Model Implementations

Implementations were performed for the CIM, IPSO, OMA LW2M data models. These involved the creation of objects, e.g. Node, Peer, Temperature Sensor, which were added to the object space as key value pairs as the node started up. Other objects, such as the Node object from a peer, could be added dynamically

after start-up on receiving the appropriate HPP *Add* message. The DM service class and instance objects were created at the start of the process, followed by the Node and Peer classes and instances and then the local instrumented objects, e.g. on the Sky mote the red/blue/green led and temperature sensor were created. These local instrumented objects and instances were mapped to the underlying Contiki code (in /dev/sensor) to get or set the actual values.

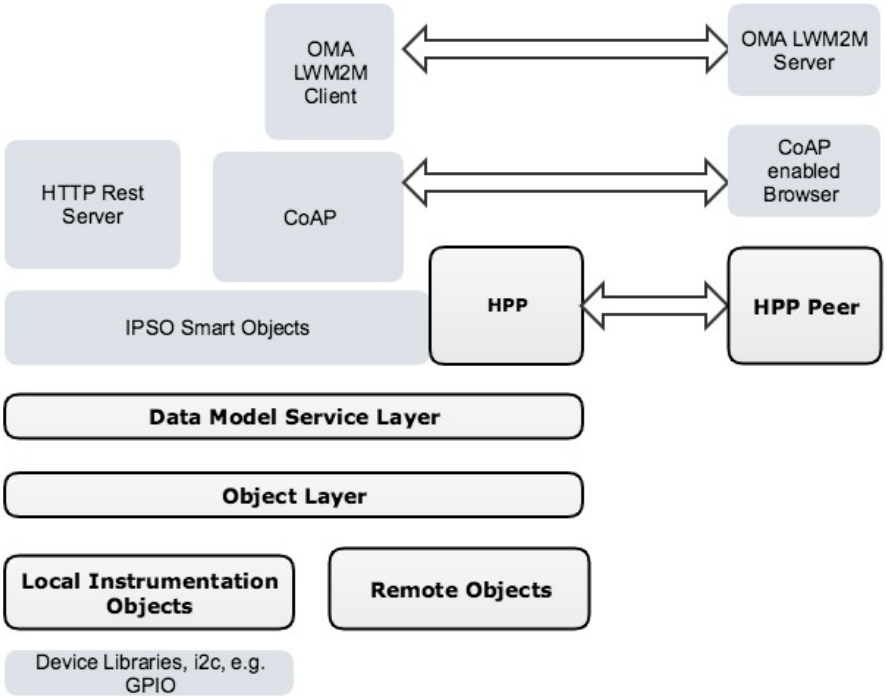

Figure 17 Resource Access over HPP and CoAP

Figure 17 shows how the Contiki CoAP implementation based on Erbium and the OMA LWM2M [148] integrated with the holistic architecture, using the data model service layer. It also shows how the same objects were accessible over HPP by a HPP peer and over CoAP, but stored only once in the object layer (and not separately for each application protocol). A separate CoAP resource was implemented for the creation and retrieval of HPP objects using a CoAP transport, i.e. a GET to /hpp/led would return the led object encoded as key value pairs, whereas a GET to /led would return the standard CoAP response. This allowed testing of the initial HPP object implementations separately to the use of the HPP protocol itself.

5.3.1 Data Model Service Layer

The data model service layer uses key value pairs to store or send objects from/to a remote node and data structures in the local instrumentation layer to encapsulate the node functions provided by the OS or node vendor libraries to access sensor data. The dm_service library provides support functions on top of the object library:

- `dm_add_class()` - for each class supported locally or from a remote node
- `dm_add_instance()` - for each local sensor or remote instance added
- `dm_remove_instance()`
- `dm_find_instance(), dm_find_class()` – to find objects using keys or particular attribute values according to the type of matching specified by parameters passed in.
- `dm_get_instance(instance_handle), dm_get_class(instance_handle)` - to retrieve an instance, using the instance handle.
- Helper functions such as `dm_li_add_class()` and `dm_add_li_instance()` were added on top of `dm_add_class()and dm_add_instance()` to make it easier to set up the li structures.

For example, a DM_SOURCE_SRV service and node objects were implemented as key value pair objects to be sent to another node acting as a DM_STORE_SRV. Classes and instances for red/blue/green leds, temperature sensor and node, using a subset of attributes from the CIM object and OMA LWM2M, were also implemented.

The following simplified code (not including error code) shows part of the start-up code where a service adds its own service class template and initialises its roles as a source of data, sink and store of data from other nodes:

```
uchar dm_register_dm_service(objectAttr_t *template_ptr,
                             objectAttr_t *inst_ptr,
                             objectAttr_t *inst_key_ptr) {

  if (dm_srv_class_hdl == 0){
    dm_srv_class_hdl = dm_add_class(NULL,
                                    &DMServiceTemplate,
                                    DM_SERVICE_CLASSNAME,
                                    NULL);

    hdl = dm_add_instance(HPP_SERVICE_LIST,
                          dm_srv_class_hdl,
                          inst_ptr, inst_key_ptr…..);
  }
  if (service_role || DM_SOURCE_SRV) {
     dm_source_init(); // initialise local
  }                    // instrumentation in object store
  if (service_role || DM_SINK_SRV) {
    dm_sink_init
  }
  if (service_role || DM_FORWARDER_SRV) {
    dm_store_forwarder_init();
  }
  return (0);
}
```

The following simplified code for HPP_ADD shows how messages carrying key value pairs are used to add a class or instance, with the receiving node calling setup_template() to process the class attributes received and then calling dm_add_class() or dm_add_instance() with the received instance:

```
if (STRMATCH(HPP_CLASS, value)) {
    void * templ_ptr;
    templ_ptr = setup_template(message);
    thisHdl = dm_add_dm_class((void*)templ_ptr, obj_name);
 } else if (STRMATCH(HPP_INSTANCE, value) ) {
    rv = processAttrElements(message); // process kv pairs
    thisHdl = dm_add_instance(classHdl, inst_ptr);
 }
```

5.3.2 Local Instrumentation Layer

This layer hides the platform specific sensor hardware implementations in order to make it easier to support the object space layer on different types of device, as

only this layer has to be ported. It also makes it easier for the port to be done to different devices of different capability as it provides the same abstractions (in the form of local instrumentation structures) on each device, which are implemented according to the device's capabilities. It provides get(), set() functions and method prototypes for node functionality to access local node data and functionality. These functions map well to the hardware/vendor specific implementations on nodes that access particular readings or data, such as an API call like get_sensor_reading() or reading a value from a register.

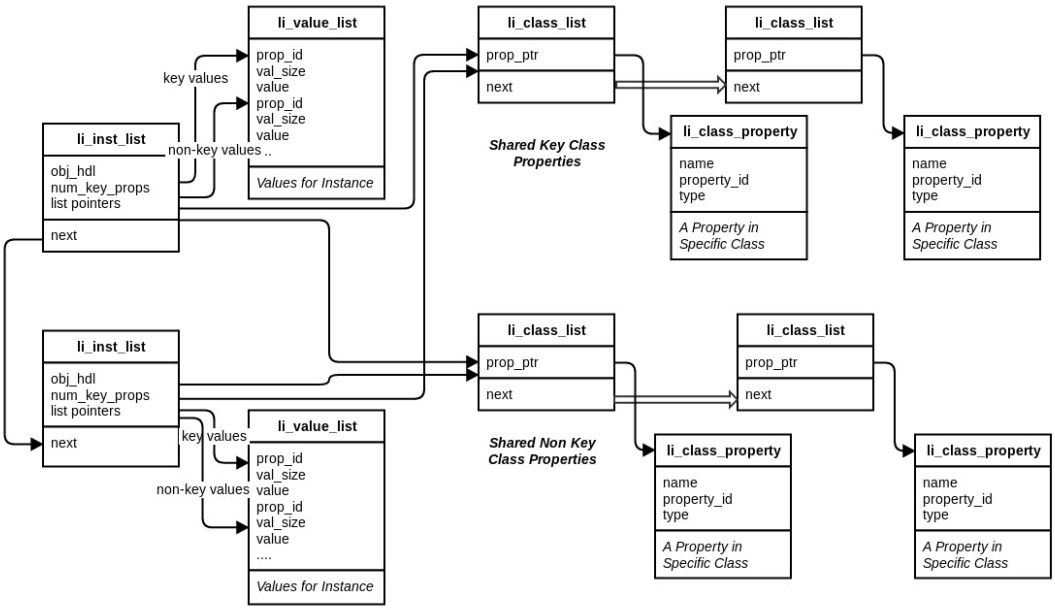

Figure 18 Local Instrumentation Structures

The local instrumentation (li) layer specifically treats each object attribute/property individually and allows these attributes to be built into objects representing only the attributes that should be implemented on a device. Figure 18 shows two instances sharing class metadata, where the description of the individual properties is stored only once in a list for the class and how the values are stored in separate lists for each instance. As well as efficient use of memory, this provides the flexibility required to map a rich data model to a resource constrained WSN device, where higher-level data models can be built up using a local instrumentation structure per attribute giving per attribute mapping to the underlying node functions or data. This also allows only those attributes supported by the node to be implemented, rather than having to store an object's unsupported

attributes. This contrasts with how objects are normally inherited with all attributes, even if not required. This approach is also very much in line with the per property (or Resource in IPSO terms) approach used in OMA LWM2M. It is straightforward to map this per property approach to a complex object such as used in the CIM information model.

The li_class consists of a list of li_class_property - one per property of the class. The li_class_property structure makes no assumptions about the object it is to be put in (it could be in several), giving the modelling flexibility outlined above.

On start-up, a node allocates and sets up an li_class structure for each class to be stored locally. It then sets up an li_instance structure for each object instance, e.g. a sensor, according to its configuration. This li_instance in turn consists of at least one li_instance_property, each of which is linked to its single li_class_property definition, so reducing memory use compared to having this in every instance.

5.3.3 Integration with Erbium CoAP Implementation

An initial integration used CoAP to carry the HPP encoded object information and message types. This was done to allow testing of the implementation of the objects in the data model service, object space and local instrumentation layers in advance of the HPP protocol itself being available. It also showed the flexibility of those layers being able to easily integrate with the existing CoAP code. The CoAP resources were accessed via URLs using a suffix of hpp/[classname] and the node responded with the properties implemented in that object as key value pairs in the CoAP payload, using multiple CoAP buffers. The code samples below show this integration with Erbium-CoAP itself was straightforward. The HPP message payload was simply added as CoAP payload using the call REST.set_response_payload(). The additional code required in Contiki compared to Linux consisted of :

- Initialisation.
  A Contiki call was added to the Contiki main PROCESS to call the initialize code in the hpp_service to set up the service and node objects.
- Integrating with the REST code.
  This consisted of code to add the resource into the erbium resource handling list `rest_activate_resource(&resource_hppnode)` and

the code to implement that resource. A RESOURCE macro is used to define a CoAP resource, the CoAP verbs such as get or put it handles and a corresponding function to implement it called resource-name_handler. The handler below for the node object returns the node instance from the object space when queried over CoAP:

```
void hppnode_handler(…) {
    object_t *instObj_ptr = NULL;
    instObj_ptr =
  dm_find_instance(NODE_CLASS);
  hpp_send_object_resp(instObj_ptr,
                       response, buffer);
}
```

- Adding a Resource for HPP Objects.
  This allowed a URI like /nodeAddr/hpp/object?hdl=x to select an object by the handle allocated when it was created in the object space or to walk through the available objects, as shown by the following handler:

```
void hppobject_handler(…) {
    len = REST.get_query_variable(request,
                                  "hdl",
                                 &chdl);
    instObj_ptr =
       dm_find_object_by_handle(hdl);
    hpp_send_object_resp(instObj_ptr,
                         response,buffer);
}
```

The following code implements a Resource to set on the led held in the hpp object space as a CIM Alarm with a CoAP POST and also shows the use of the local instrumentation (li) layer's li_mote_method() to call the underlying hardware to make the setting on the LED device.

```
RESOURCE(hppsetled, METHOD_POST | METHOD_PUT ,
"hpp/led/set", "title=\"ledset\";rt=\"Control\"");

void hppsetled_handler(void* request,
                   void* response,
                   uint8_t *buffer, uint16_t
                   preferred_size,
                   int32_t *offset) {

  int rv = li_mote_method(MOTE_CAP_LED_SET,
                          LEDS_RED, MOTE_LED_ON);
}
```

- Integrating with the Contiki hardware abstractions using the local instrumentation (li) layer.
  This simplified code shows the li layer code wrapping the Contiki led calls and is called by a resource handler to set a led:

```
li_mote_method(int method_cap, int
               inst_id,
               int setting) {
uint8_t led = (uint8_t)inst_id;

if (method_cap == MOTE_CAP_LED_SET)
   if (setting == MOTE_LED_ON)
       leds_on(led); // Removed leds_off, leds_toggle
}
```

5.3.4 Contiki Implementation of OMA LWM2M

This implementation created objects in the object space using the data model service layer and these were available directly over HPP and over the existing REST interfaces. Use of the existing REST interfaces made testing easier by allowing the use of the Copper Browser plugin and Leshan tool for OMA. It also allowed the flexibility of the holistic architecture to be considered in terms of its ability to support protocols other than HPP.

Existing OMA LWM2M Contiki Code

The existing OMA LWM2M ipso-example implementation in Contiki3.0 creates a context for parsing a CoAP request. It uses a set of structures to represent objects and resources, with enumerations for types of resources. The LWM2M object

structure contains pointers to instances, which in turn contains pointers resources as below:

```
typedef struct lwm2m_object {
    uint16_t id;
    uint16_t count;
    const char *path;
    resource_t *CoAP_resource;
    lwm2m_instance_t *instances;
} lwm2m_object
```

The ipso-example code initialises the LWM2M engine in a thread that calls:

1. `lwm2m_engine_init()`
2. `lwm2m_engine_register_default_objects()` to set up a device object for the node
3. `ipso_objects_init()` to initialise the supported objects, e.g. `ipso_temperature_init()`.
4. enter a loop processing events.

The object init() methods code for the temperature resource is shown below , with other resources having similar code. This code segment shows the Object Id, type and value as defined for this LWM2M resource (and used to form its URI as shown earlier). The call-back will be a method to access the real values. Note that this method call to access the real values is specific to this implementation, unlike the more generic approach using the local instrumentation layer in the holistic architecture.

```
LWM2M_RESOURCES(temperature_resources,
   LWM2M_RESOURCE_CALLBACK(5700,{temp, NULL, NULL}),
   LWM2M_RESOURCE_STRING(5701,"Celcius"),
    // some entries not shown…….    .
   LWM2M_RESOURCE_FLOATFIX_VAR(5602, &max_temp));
```

An instance will be created and included in an object by:

```
LWM2M_INSTANCES(temperature_instances,
   LWM2M_INSTANCE(0,temperature_resources));
   LWM2M_OBJECT(temperature,3303,
                temperature_instances);
```

This is followed by a call to add this object to the engine's static array of lwm2m_object_t pointers -

```
lwm2m_engine_register_object(&temperature);
```

Mapping to the Holistic Architecture – Setting up the IPSO Objects

The process of mapping the LWM2M code to the holistic architecture consisted of two main items. The first item was to set up the IPSO objects for storage. The second was to interface with the LWM2M engine and its CoAP processing code and to replace the code it used to access the stored data or the underlying hardware with code using the data model service layer, e.g. to read a temperature sensor.

Setting up the IPSO objects began by creating a set of header files with static definitions for IPSO resources (properties in HPP) and objects, e.g.

```
#define IPSO_Sensor_Value_PROP_ID 5700
#define IPSO_Generic_Sensor_OBJECT_ID 3300
```

Then a static definition of the IPSO Classes was created in a header file, initialised as an array of li_class_property to hold the property names and types of the class, as per Figure 18. For example, a Temperature Sensor class is defined as:

```
  //name,property_id,type, mode, permission

  li_class_property_t IPSO_Sensor_Value = {
        IPSO_Sensor_Value_PROP_NAME,
        IPSO_Sensor_Value_PROP_ID,
        real32, DYNAMIC, READONLY};
```

These properties are then grouped into statically defined li_objects with valuelists of key value pairs (or callbacks to set values) as in:

```
li_kv_entry_t tSensor_vals[] = {
  // property id, length, value, next
  {IPSO_Sensor_Value_PROP_ID,4,"0",
               &tSensor_vals[1]},
  // some entries not shown…….   .
  {IPSO_Sensor_Type_PROP_ID,12,"Temperature",
               &tSensor_vals[9]},
  {LI_END_PROP_ID, 0, NULL, NULL}
};
```

Finally, the classes implemented in a given node are added to an array of li_class_t to define the properties (by pointing to that list) with the relevant callbacks:

```
li_class_t node_classes[] = {
         IPSO_Generic_Sensor_OBJECT_ID,
         HPP_PREFIX,
         IPSO_GenericSensor_PropCount,
         &IPSO_GenericSensor[0],
         0, NULL,&localFunctions},.......};
```

For dynamic properties, such as the reading taken from a sensor, the getter/setter callbacks per property must be coded to access the values from the hardware.

Mapping to the Holistic Architecture –Interface with the LWM2M Engine

Having defined the properties in the local instrumentation structures, the definitions for those attributes to be instantiated on a given node, based on its capabilities (such as sensors on board and its memory size), have to be integrated into the existing LWM2M engine. The LWM2M context and REST code were retained and the `lwm2m_engine_register_default_objects()` method was extended to call `dm_service_initialise`(). This call sets up a DM_SOURCE_SRV with `service_source_init()`. This call initialises the li_node information and calls `dm_li_add_class()` and `dm_add_li_instance()` to add the supported DM service, node and local instrumentation classes and instances to the object space. For example, a local instrumentation instance such as a Generic Sensor is added with a call as below:

```
  rv = add_inst(this_info_ptr,
                &myGenericSensorInstances[0],
                IPSO_Generic_Sensor_OBJECT_ID,
                "0",
                IPSO_Keys_PropCount,
                IPSO_Generic_Sensor_INDEX);
```

,

Finally, `lwm2m_engine_handler()` was changed to use data model service layer calls such as `dm_find_instance_by_name()` to get and set LWM2M resources, returning the appropriate REST responses as before.

5.4 HBase Integration

To demonstrate the effectiveness of the abstractions to model the end-to-end data flow, a DM_SINK_SRV service was written in Java to integrate with HBase.

HBase was chosen as an example of a Big Data NoSQL store to be used with sensor data, because its columnar nature and dynamic schema suit the variety of sensor data which may be received in structured, semi-structured or unstructured formats. This service used the CIM information model for sensor data, which is then stored in HBase. As discussed in section 2.5, the verbose nature of the CIM information model is not well suited to use on WSN devices, but it models sensors well, with a rich information model making it a useful information model to demonstrate (or not) the usefulness of the holistic abstractions in interfacing with a datastore such as HBase.

A DM_SINK_SRV service was created to write to HBase, as shown earlier in Figure 15. The model used was to create a HBase table for each HPP class (on receiving a HPP *Add* Class message) with a row for each instance and column families for "key attributes" and "attributes" (stored separately in the hpp object), with a column family qualifier for each attribute. A row key consists of the object's key attributes, node identifier and a timestamp. The code extract below shows the writeToHBase() method. It assumes the table has already been created by an earlier HPP *Add* command and shows how the received key value pairs in the HPP data are processed and written as a row to the HBase table for that class.

```
// required try catch blocks not shown
public static void writeToHBase(Configuration conf,
                                String tableName,
                                String hppData) {

    Map<String, String> keyKvs = getKeyMap(hppData);
    Map<String, String> attrKvs = getAttrMap(hppData);

    HBase admin = new HBaseAdmin(conf);
    HTable table = new HTable(conf, tableName);
    String rowKey = createRowKey(keyKvs);
    Put put = new Put(Bytes.toBytes(rowKey));
    // Add hpp data to column families
    addMapToHBasePut(put, keyKvs, "key attributes");
    addMapToHBasePut(put, attrKvs, "attributes");
    table.put(put);
    admin.close();
}
```

A method was also written that allowed a HBase scan of a given column/column family in a particular table.

This DM_SINK_SRV service writing to HBase could act as a HPP peer and join a peer-to-peer network, as shown in earlier 'C' code in section 5.2.3. The flexibility of the holistic abstractions also allows the HPP object to be carried over CoAP.

```
public static  boolean createRecord(String addr,
                          String method, String url,
                          String payload, int timeout)
                                    throws IOException {

    Configuration conf = HBaseConfiguration.create();
    COAPPacket request = new COAPPacket(draft);
    // Code to set header, method, payload not shown
    DatagramPacket reply = send(addr,
                        port,
                        COAPPacket.serialize(request),
                        timeout);
    if (reply != null) {
        COAPPacket pkt = new COAPPacket(draft,
                                        reply.getData(),
                                        reply.getLength());
       writeToHBase(conf, pkt.getPayload());
    }
}
```

The code above for createRecord() shows this use of CoAP, where the DM_SINK_SRV acts as a Java CoAP client. That client connected to the desired WSN node via a socket to the CoAP Server on the Contiki RPL border router. As per the code segment above, it builds a COAPPacket using COAPPacket(), calls the serialize() method and sends the request using the CoAP libraries. On receiving the reply, containing the HPP data, it writes that data to the HBaseConfiguration object it had created to writeToHBase().

5.5 CacheL Implementation

The CacheL algorithm is outlined in section 4.2 with simplified 'C' code to illustrate its operation. That code shows how the implementation uses some specific metadata (such as the last time it was visited on a sweep and its access count) stored in the cached items. This is done in order to avoid having to update all entries in the cache on every sweep. For example, the number of times the access count missed being decremented in a sweep (due to the periodic nature of

sweeps) is handled by holding a sweep count which is incremented on each sweep, which is then used to decrement the access count on the next sweep.

CacheL was implemented in ‘C’ and was incorporated into the object space described earlier to manage the replacement of objects in the object space using lease and access count. In this instance, the commonality of code and metadata for cache and lease management was designed to reduce code size and memory use, as well as making the code simpler.

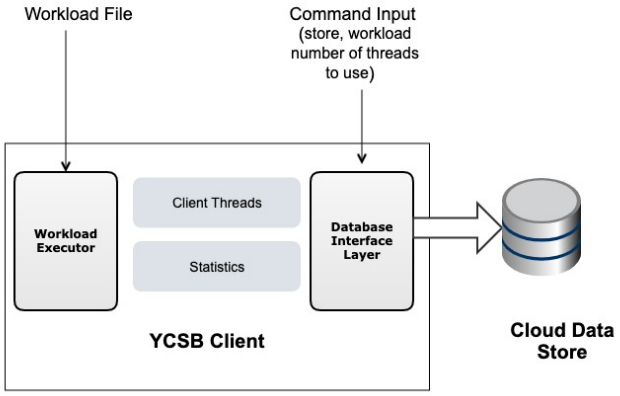

Figure 19 YCSB Test Setup

CacheL was also implemented in Java so that it could be compared to a Java LRU implementation using the Yahoo Cloud Server Benchmark (YCSB). . Figure 19 illustrates the test setup used for YCSB. The Java LRU implementation used HashMap's removeEldestEntry() method. The CacheL implementation also used a HashMap to be comparable, although this needed extra code to handle ConcurrentModificationExceptions by marking items for deletion in a sweep and to remove them later. It also had to hold a separate list of index-object reference pairs so a sweep could continue from the previous position, which the C implementation did not need as it used its own circular list library.

YCSB was chosen for a number of reasons. Firstly, it allowed the use of larger data sets than would be typical on a WSN node in order to determine the effectiveness of the algorithm in a non-constrained environment. Secondly, YCSB also provided a test infrastructure that was relatively easy to use and flexible in terms of adding data sets with new distributions and being able to run workloads

that varied the read and update ratios on those data sets. In its normal use case, YCSB runs separate commands to load a database and then to run a set of tests on that data. In this use case of an in-memory cache, these runs had to be combined. Furthermore, the update implementation was not a simple insert as with databases, as the item to be updated may not be cached and so had to be read and inserted.

5.6 Summary

This chapter has given an overview of the prototype implementations for both Linux and Contiki3.0, which were performed in order to realise and evaluate the components in the holistic architecture. It is important to note that the 'C' implementations allowed re-use of large parts of the same codebase to run on the constrained nodes and more capable Linux nodes, as per the design goal for the architecture. In particular, this chapter has described implementations of the following:

- the layers in the holistic architecture
- the CacheL algorithm
- integration with the OMA LWM2M stack and Erbium CoAP stack
- OMA LWM2M and CIM information models, including the per attribute mapping and the effectiveness of the local instrumentation layer in its use of memory
- a DM_SINK_SRV accepting HPP objects and storing them in HBase

The next chapter presents the results of a set of tests on the implementations described in this chapter. It also presents an evaluation of the components in the architecture based on those results. This includes quantitative analysis of the size of the code and the hit ratio of CacheL versus LRU, as well as scalability tests and qualitative assessments as to whether the abstractions made it easier to develop software for constrained node devices and to assess the portability of that software.

6 Experimental Evaluation

This chapter presents the results from implementing the layers of the holistic architecture and HPP protocol. It considers the success of the architectural abstractions against the requirements and considers its use in some example scenarios. This is followed by qualitative consideration of complexity and quantitative consideration of HPP and CacheL, such as performance and memory use (to determine the feasibility of its use on constrained devices).

Sections 6.1 and 6.2 evaluate the results in broad terms of whether the requirements in each section were met and how the holistic architecture compares to the other architectures presented in earlier chapters. Section 6.3 shows a mapping of the holistic architecture to some example scenarios. Section 6.4 gives qualitative assessments of the key issues for developers on constrained devices, i.e. implementation complexity and APIs (**Req-4**). Sections 6.5 and 6.6 consider the requirements more specifically by considering the success of the abstractions in modelling the interactions involved (**Req-1, Req-2, Req-6**), the success of mapping data models into the object space and local instrumentation layers (**Req-3 and Req-5**) and the success of the P2P approach using DHT in a decentralised manner (**Req-7**). Section 6.7 considers the memory use of particular data model implementations on Contiki compared to other libraries on Contiki. Section 6.8 considers the performance of HPP in a number of test scenarios on both Contiki and Linux in terms of scalability and robustness. Section 6.9 provides a qualitative consideration of the complexity of implementing CacheL and quantitative evaluations for the hit/miss ratio of CacheL and optimisations introduced during testing, as well as for the memory use **(Req-8)** of code and data.

6.1 Consideration of Architectural Requirements

The architectural requirements from section 4.1.2, and how the holistic architecture addresses them, are summarised below:

- *Req-1. Clearly define the possible roles of nodes*. This is explicitly catered for by the roles defined in the data model service layer, e.g. DM_STORE_SRV. It is further supported by the exchange of the HPP messages in the peer's capabilities in the *Hello* message and the roles in the service object.

- *Req-2. Provide abstractions for the basic operations required of a sensor node and the services, which map easily to a range of heterogeneous devices and higher-level services*. This is provided by the local instrumentation layer providing the device specific low-level functionality in a way that can be exposed to the object space and data model service layer, which is the common way to access the data held on the device and its functionality.

- *Req-3. It must be independent of particular node hardware and must handle a range of node functional capabilities*. The data model service layer and object space are independent of the node hardware and are able to hold data to support a range of node functions with the local instrumentation layer providing the implementation of node specific functions.

- *Req-4. It must provide simple, consistent APIs for developers of device and application software*. The object space provides a simple and consistent API for both the data model service layer and local instrumentation layer, as well as higher level applications. The data model service layer also provides a simple API to application developers to manage and access the data stored and the lower layer functionality. Furthermore, the commands in the HPP protocol align well with these APIs.

- *Req-5. It must provide a consistent means to exchange sensor information independent of the underlying technology and provide specific support for the modelling of sensor data to allow integration into higher level systems*. The IPSO and CIM data models were successfully implemented, as well as an integration with the OMA LWM2M stack and the Erbium CoAP stack. A mapping to the OpenFog reference architecture is also shown.

• *Req-6. It should support a sensor node informing other nodes and services of its sensing and platform capabilities.* The platform capabilities in terms of messages supported are exchanged in the *Hello* message as well as the HPP version and the sharing of the service object using *Add* or *Get*. The sensing capabilities can be determined by discovering the supported objects.

• *Req-7. It must be able to handle small, static networks and allow the system to adapt as the network grows/changes or encounters other networks, supporting applications discovering and collaborating without a centralized coordination facility*. The use of an overlay network in HPP with a DHT makes it able to identify and exchange data in a network and across a variety of networks, using an identifier (a prefix part of which may be specific to a given network) without centralized coordination. The use of a lease also supports the dynamic nature of nodes joining or leaving more easily.

• *Req-8. It must use protocols that are sufficiently simple for low capability devices to participate, according to their capabilities*. The holistic architecture and HPP have been implemented on constrained devices such as the TMote and WiSMote.

6.2 Architectural Comparison

In terms of the OpenFog Reference Architecture, the local instrumentation layer and the object space layer align with the Node view of the Sensor and Actuator layer and the Protocol Abstraction layer. The layers in the holistic architecture, however, provide a richer set of abstractions and detail than the lower layers in the OpenFog architecture and the holistic architecture retains this consistency by using the abstractions on higher level systems. The defined data model service layer and its service roles fit the storage functionality in OpenFog and the holistic abstractions could be useful for the layers to manage nodes and for the Application layers. As such, HPP allows fog and edge computing components to be interoperable at the level of providers and architecture models and interfaces. HPP also provides a means to exchange information between fog nodes and Cloud services (an example integration with a typical HBase as a typical Cloud NoSQL database is shown in section 5.4).

In terms of the RESTFul Style constraints given in section 3.2.1, it can be seen that the RESTFul use of a well-known URI for discovery is replaced by the use of a *Hello* message and a node having to know one node address to join a peer network. The limited and consistent set of message types in the HPP protocol aligns with the RESTFul use of methods with the same semantics for all resources. This is strengthened beyond REST in the holistic architecture as the same semantics also map to the object space API. The use of HPP to manipulate objects aligns with the RESTFul manipulation of resources through the exchange of representations and its use of self-descriptive messages to exchange representations.

The comparison of functionality of the HPP protocol with CoAP is similar to that above of the holistic architecture with the RESTFul architectural style, as they both reflect the architecture they were intended for. In particular, the HPP commands cover equivalent functionality to the CoAP verbs, although *Take* is richer than DELETE and *Notify* is implemented differently to OBSERVE. There is no explicit call to a resource's methods in CoAP and this is usually achieved by setting/resetting an attribute in the object, whereas HPP allows a call to the method defined in the object.

While both CoAP and HPP provide for discovery, HPP does not rely on a centralised server as with resource discovery in CoAP and HPP does not rely on every server providing the /.well-known/core endpoint for the Resource Directory(RD) or other nodes to discover its resources. Furthermore, CoAP does not specify how a node announces itself on joining the network, whereas a HPP peer can discover other peers and not just the resources on a node, once one well-known peer identifier is known.

It can also be said that the capabilities exchanged in the HPP *Hello* and the roles in the service object provide a richer view than the Resource view available over CoAP. This also allows HPP nodes to act only as sources and connect outwards only, whereas the use of CoAP in OMA LWM2M requires accepting and making connections. Furthermore, while CoAP supports caching, the use of a lease in the holistic architecture's cache and explicit support for setting and renewing leases in HPP reflects its more central role in HPP. The current prototype HPP implementation does not have a binary encoding and reduced headers as provided by CoAP, but it is expected that the design of HPP will allow this in future.

The approach taken in the WoT work is similar to the approach taken here in that it identified requirements and designed an abstract architecture, based on those requirements, with interoperability and support for multiple information models at

its core. Hence, it can be said that the holistic architecture shares a data-centric view and the RESTFul Architectural style constraints with WoT. The holistic architecture is, however, not specific to web technologies. HPP could interoperate in the Web of things, but also provides HPP as a protocol, a richer set of roles and architectural layers that can be used as appropriate on a node. It also does not need a separate thing directory to contact a local device as in WoT, as this is achieved by the HPP DHT. The holistic architecture is likely to be less demanding of resources than WoT as the suitability of the WoT for constrained devices is to be determined.

Section 2.6 also showed that the WoT approach to interoperability is to provide interaction across platforms by supporting an infrastructure for IoT platforms using different protocols to communicate and use the thing Description (TD) and an exposed thing for this TD as the shared information at a Web application layer. This use of the TD is similar to HPP providing capabilities, although HPP expresses these as HPP commands and service roles, with the information model being a separate object that could be returned as part of the node object or indeed the TD itself could be retrieved by a HPP *Get* message. WoT is prescriptive in the use of a Thing Description used to build an instance of an exposed client. The holistic approach in this thesis is less so and includes a simple *Hello* that allows a peer to participate. Furthermore, as the holistic architecture uses an overlay by design, it does not have the richness of proxy and bridge capabilities found in WoT.

6.3 Example Scenarios

The holistic architecture and prototype implementations of it have been presented in earlier chapters. Figure 15 showed how the layers within the holistic architecture interact according to the capability of nodes and Figure 17 showed how resources could be accessed over HPP and CoAP, while sharing the object space to only store the underlying data once. This section illustrates how these generic capabilities could be used in real scenarios.

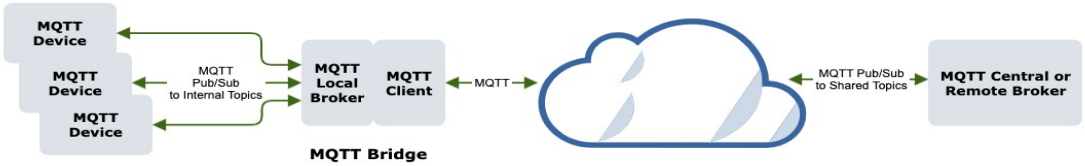

Figure 20 MQTT Bridging

Figure 20 shows an example where MQTT brokers are federated using the current MQTT bridging approach. Each bridge has to be configured with the address and port of the remote broker or brokers, a client name, the topics the broker will publish and the topics it will subscribe to, along with local and remote prefixes. As can be seen, this involves a high degree of manual configuration and the federation is done in a manner specific to MQTT.

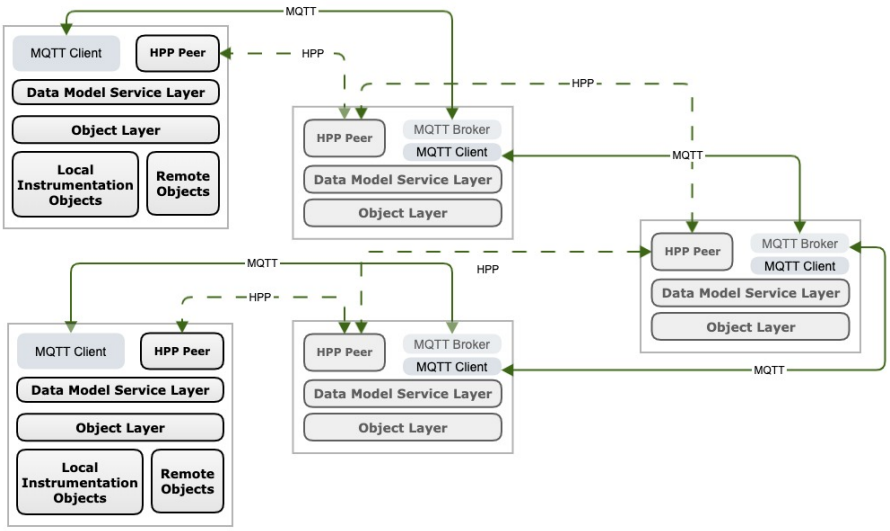

Figure 21 Federation of MQTT using HPP

Figure 21 shows how the holistic architecture could realise this scenario by HPP peers. HPP could be used to find the IP addresses of the relevant peers with MQTT brokers. One approach could be for those peers sharing a node with MQTT to either *Add,* or respond to a *Get* for, an object outlining the MQTT configuration such as topics over HPP and this could be used to create a local

configuration file for MQTT. Another approach would be for a peer to send an *Add* to a specific info_hash for a group (like a BitTorrent swarm) of peers with an MQTT object matching the topic(s), as described in 4.5. Both of these examples show a more scalable, autonomous approach than the manual configuration for MQTT bridging. Furthermore, these HPP approaches could be used to federate other systems.

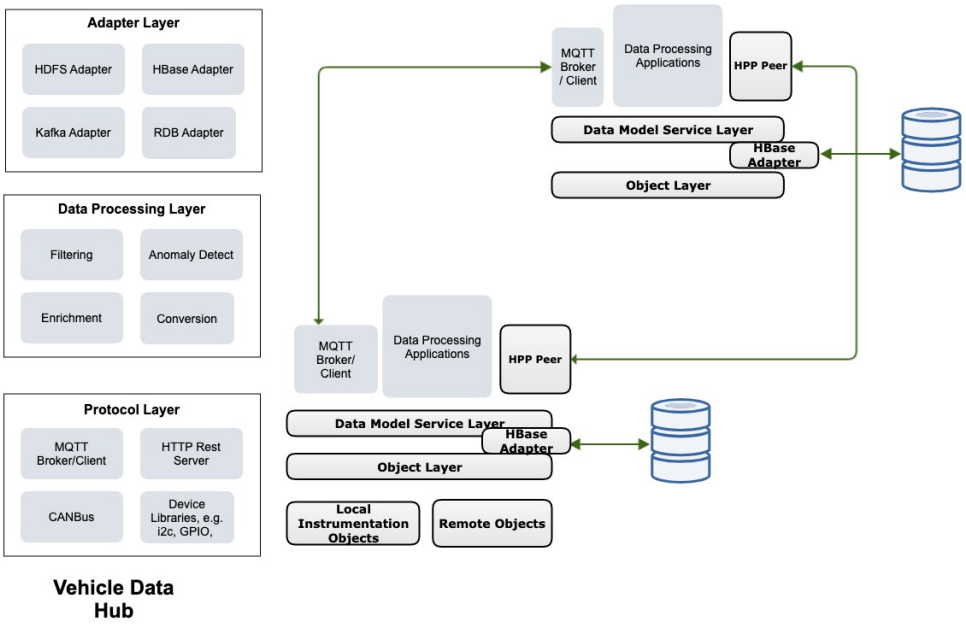

Figure 22 Vehicle Data Hub Architecture

MQTT may be part of a larger instrumentation and control system, such as in a car. Figure 22 shows a simplified view of a vehicle data hub, based on one provided by IBM [151] and it also shows how the holistic architecture's layers could map to elements in that hub. The protocol layer in the vehicle data hub maps to the holistic architecture's layers to handle local instrumentation, store it in the object space and make available over HPP or MQTT using the Data Model Service Layer. It also shows the vehicle data hub's adapter layer mapping to the holistic architecture, as per the integration with HBase in 5.4. Other adapters such as for Kafka, HDFS and relational databases would integrate with the Data Model Service Layer similarly. The vehicle data hub's data processing layer would be implemented as a set of applications on top of the Data Model Service Layer using its API. Figure 22 shows the flexibility of the holistic architecture, where the layers could be implemented on a single hub or distributed across multiple

hubs (where the adapter layer could reside in the cloud) using either the existing federation approach of MQTT or using HPP as above.

Figure 23 from the OpenFog consortium illustrates a smart transportation system scenario consisting of multiple different entities, such as low level sensors, gateway nodes and cloud based services as an example of fog computing.

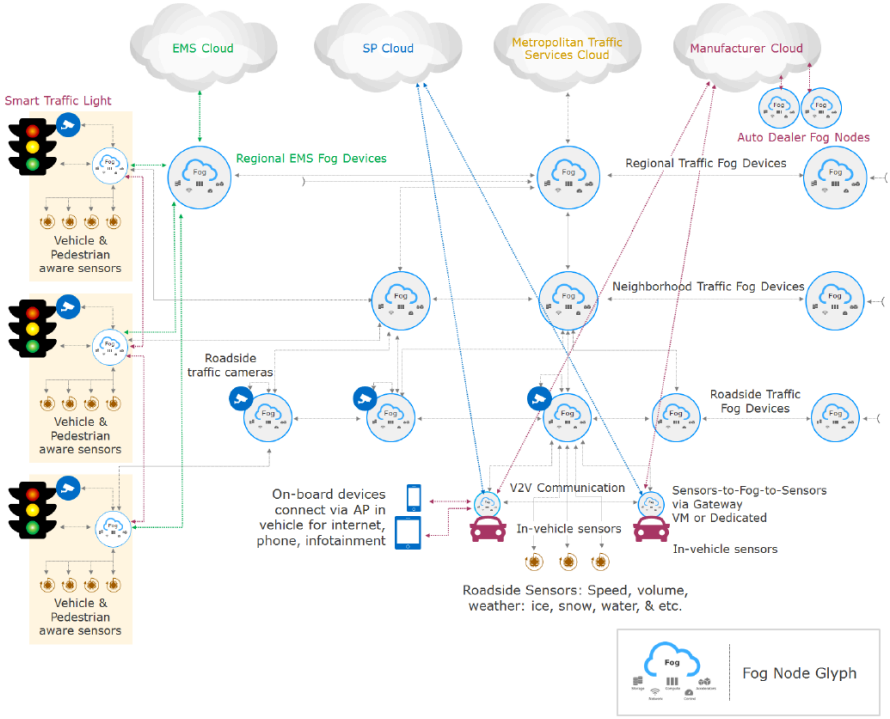

Figure 23 OpenFog Transportation:Smart Car & Traffic Control System [96]

Figure 24 shows how that smart transportation scenario could be implemented using the layers defined in the holistic architecture for both the on-vehicle and vehicle-to-cloud aspects (where the HPP communication between peers is equivalent to that between fog nodes). In this scenario, the high volume, high velocity camera data is simply forwarded to another peer and the main benefit of the holistic architecture is for nodes to discover each other and their capabilities (such as a camera feed represented by an object), with the object space used to hold information on other peers and HPP used to carry the lower volume, lower velocity information about peers and their functionality.

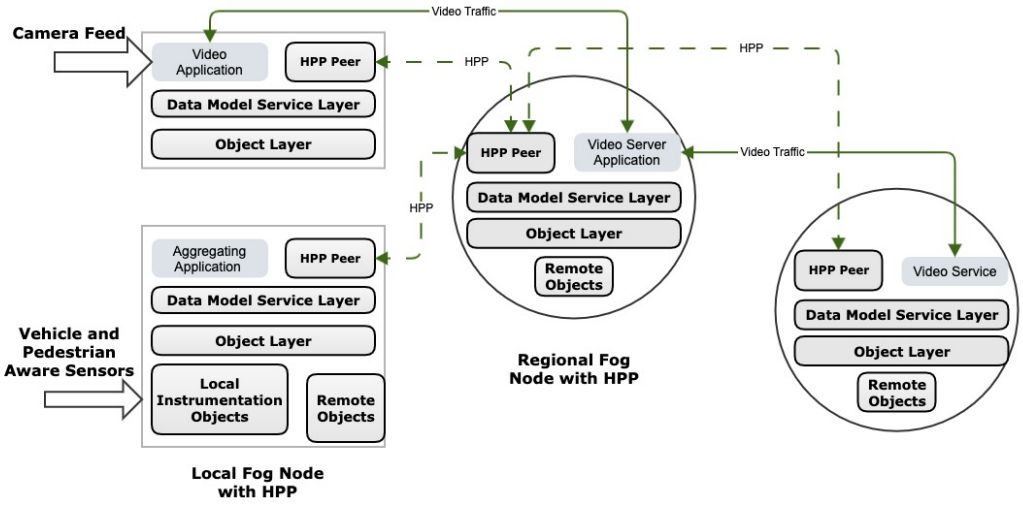

Figure 24 Smart Transport Fog System with HPP

It is important to note that even when HPP exists with other services such as MQTT or high volume video feeds in these scenarios, the holistic architecture still provides a means for lower volume, lower velocity traffic to identify peers, exchange information, discover and share roles and capabilities and to store instrumentation data once (with caching support) or to interface with richer Big Data stores for higher volume data.

6.4 Linux and Contiki Implementation

The approach of initially implementing the HPP layers on Linux allowed the design to be refined and the code to be debugged and tested more easily and rapidly, using the more advanced Linux development and debugging environments. It also provided implementations for services on Linux that could integrate easily with those on constrained nodes as envisaged in the end-to-end flow of data. The Linux services also allowed easier and quicker testing of a large number of devices, particularly in terms of testing the HPP protocol.

These benefits of a Linux implementation came at little cost in terms of the subsequent port to Contiki as most of the code did not require any recoding, given the availability of standard C libraries in Contiki. The main code changes were to provide a separate socket handler on Contiki, a revised Makefile and a simplified implementation of gettimeofday() for object leases. The second code extract in

section 5.2.1 shows the separate socket code required for handling the event-based model of Contiki's sockets. That code also shows that the channel and endpoint abstractions did make it simpler to create that socket handling code and integrate into the rest of the HPP processing code.

Another set of changes was required to improve memory use and support static memory allocation on Contiki to avoid runtime heap and stack issues. As shown in the code extracts in section 5.2, macros such as NEW_V and methods such as hpp_str_cpy were used and these used dynamic memory allocation on Linux, but fixed size memory on Contiki. It also involved changing structure members to reduce size (e.g. from int to char). This allowed the application developer to call the same library functions on both Linux and Contiki. In order to avoid stack overflow, care was taken to reduce the number of parameters passed in function calls and to change the type of function parameters, e.g. from int to char.

The code extracts for HPP implementation in section 5.2 show that it was relatively straightforward to implement the code to process HPP messages and interact with the holistic architecture's layers to store and manage data and also to incorporate the DHT implementation into processing HPP, particularly with the *Hello* and *Hello reply* messages.

The code extracts for implementing support for the data models in section 5.3 showed that the data model service and local instrumentation layers integrated well with the object space layer, as the data model service layer provided a simple API to incorporate data models and the object space and the local instrumentation layer provided a nice abstraction for lower level developers to add the code to handle device specific aspects, e.g. to access a register for a reading. The code extracts in section 5.3.3 show the straightforward nature of the integration of Erbium's CoAP implementation. 5.3.4 shows that mapping the IPSO resources to the holistic architecture's object classes and instances was straightforward. It also shows that it was simple to integrate the OMA Engine with the data model service layer and object space without requiring specific knowledge of the device being used, which was only needed in the local instrumentation layer if accessing hardware or OS functionality.

6.5 Use of Abstractions

The usefulness of the abstractions provided specific benefits for the mapping to particular data models, shown by the ease of incorporating a DHT and also CacheL into the object space. These cases are considered in later sections, but this section considers the abstractions from a user's perspective. One approach to evaluating abstractions is to consider the "end-user" and "WSN geek" [152], where the "end-user" is a domain expert concerned with using the WSN data, while the "WSN geek" is concerned with lower level WSN details such as network/node specific implementations. In this regard, this work has shown successful examples of implementations of different services to run on different hardware and interact using the same layers and base code.

The "end-user" is able to access the sensor data simply with known CoAP Resources or HPP messages from a DM_SOURCE_SRV on a constrained node or from a Java DM_SINK_SRV running as a HBase store (see HBase Integration below). This availability of information was supported by a DHT that was incorporated into the HPP protocol implementation easily using channels and the object space.

The "WSN geek" has been provided with an architecture using the object space, data model service and local instrumentation layers for incorporating node specific functionality and capabilities. The code extracts show that these items made it straightforward for a node to implement objects from a rich information model on both a Linux and Contiki platform and to map to CoAP Resources (see section 6.5.1). The code extracts also show how the channel and endpoint allowed the creation of library methods by a "WSN geek" that could hide the specifics of implementation on Contiki, such as Contiki's socket handling or the care needed with memory allocation. Hence, the abstractions for channels, endpoints and the data model service layer allowed the higher layer HPP message handling and services code to be unchanged in both environments for the application developers.

For the "WSN geek", the value of the object store's non-prescriptive nature in holding classes and instances was also shown in the implementation, where it was flexible enough to store objects from both local and remote nodes, as well as

storing DHT data. This non-prescriptive nature also allows different data models to be provided for “end-users”. It also allows an “end-user”, who is a developer to only require knowledge of one simple API, where the use of a lease also simplifies the management of the constrained storage for him.

For both the “end user” and “WSN geek”, the design goal of the same abstractions giving a generic information infrastructure across heterogeneous platforms of different capability was met and the data was able to be retrieved using protocols other than the HPP protocol.

6.5.1 Data Models

The value of the holistic architecture layers was also shown in the implementation and the code extracts earlier. For example, the object space layer’s ability to define object instances and classes on a per attribute basis also provided the flexibility required to map a rich data model to a resource constrained WSN device. This meant higher-level data models could be built up using a local instrumentation structure per attribute giving per attribute mapping to the underlying node functions or data. This allows only those attributes supported by the node to be implemented, rather than having to store an object's unsupported attributes. This contrasts with how objects are normally inherited with all attributes, even if not required. This approach is also very much in line with the per property (or Resource in IPSO terms) approach used in OMA LWM2M and this per property approach also mapped easily to a complex object such as used in CIM. The existing LWM2M implementation on Contiki3.0 cleanly maps the object, instance and resource concepts in LWM2M using a set of structures. Unlike the HPP dynamic approach to adding objects, the existing LWM2M does, however, use a static array to hold pointers to the instances, albeit this is hidden by methods like *lwm2m_engine_register_object()*. Similarly, the implementation of the object space, is hidden by the object library API.

The value of the data model service layer’s abstractions for roles was shown by the implementation of the DM_SOURCE_SRV role for the LWM2M objects. This integration shows that a more complete integration of the HPP protocol with the IPSO and OMA code could take advantage of the ability of the proposed holistic architecture to store and cache data from remote nodes by using the

DM_STORE_SRV role for LWM2M data from remote nodes. LWM2M does have the concept of registration to one or more servers, which includes objects, but this does not appear to be as rich as the defined data model service roles. The use of the data model service layer also allows a much richer matching in a request than OMA LWM2M as it can match on template or wildcards or particular properties.

Also, the data model service layer distinguishes key and non-key properties in the class, allowing it to handle the LWM2M use of a URI of objectid/instance or objectid/instance/resourceid to select a resource, but also to be flexible enough to support the straightforward implementation of other data models, e.g. the keys used in CIM.

6.5.2 HBase Integration

The architecture allowed data on the WSN node to be transported and stored in HBase, using CoAP or HPP, requiring no application level proxy and only requiring a proxy at the network level, i.e. the RPL border gateway. The HPP message types also mapped well to HBase functionality. For example, the two column families defined for attributes allowed a HPP *Add* of a class template to dynamically create new objects with their attributes by creating a table (and its columns). Subsequently, a HPP *Add* of an instance will then result in a new row in the object's table.

In terms of data mapping, the HPP objects mapped cleanly to HBase tables and the approach of composing an object from individual attributes mapped well to HBase columns. Furthermore, the approach of separate key and non-key properties also mapped well to separate HBase column families, allowing a HBase scan across all rows of key attributes as well as non-key attributes, rather than only being able to use the key attributes as instance identifiers.

6.6 Mapping of OMA LWM2M and CIM Data Models

The implementation of the CIM and OMA LWM2M models both required mapping of their respective object models into the object space. It can be seen that the per property(resource) data model of LWM2M is more suited to constrained

devices, e.g. the IPSO Generic Sensor definition is much simpler than the inheritance involved in constructing a CIM_NumericSensor. CIM also uses lots of strings, e.g. for names, which is expensive in memory, even if only stored once as in HPP, whereas the ids in IPSO are easier to program and more efficient in memory. IPSO also has fewer types than CIM, as suits constrained devices. Using digits for object identifiers reduces string usage and is suitable for M2M, but it is more user friendly to use a RESTFul well-known URI, so names were also supported in the implementation, e.g. /Device/0/Manufacturer as well as 3/0/0.

CIM does, however, provide a clearer model regarding methods as it has specific object methods, whereas IPSO uses resources with implied actions, e.g. CIM has setAlarmState which can be used to set a led, whereas IPSO Light Control uses the On/Off (5850) boolean Actuator resource. In the case of an LED, the IPSO approach is simpler and reasonably obvious, but it is less obvious in resources like "Reset Min and Max Measured Values", while the implied use of "On-Time"(5852) or "Off-Time"(5853) to reset is not consistent with specific Reset resources elsewhere.

The implementation of the OMA LWM2M and the IPSO Objects has shown that a per property based approach fits naturally with how the low level functionality is often performed on devices, e.g. with a GPIO call per property and maps to CoAP REST resources, such as led and sensors and groupings of individual attributes. It also allows selection of only the implementable attributes on a node, so saving memory per implemented class. Both LWM2M and HPP support this approach. While REST resources such as led and sensors generally have a few properties, the IPSO Application Framework [153] defines function sets as groupings of individual attributes, e.g. a device at /dev has 12 resources, e.g. Manufacturer at /dev/mfg.

This per property design approach is implicit in IPSO's objects, but CIM is intended to be implemented as a full object model. This implementation of a subset of several CIM object classes (with many strings) has shown that even the CIM_Sensor is too heavy for implementation on a resource constrained device, as it has over 20 possible attributes. The local instrumentation layer and the class template with attribute descriptions and its instance object with attribute values allowed a selective per property approach that meant only those attributes

available on the constrained node hardware were coded and it supported a set of abstractions in a COAP/REST environment that allowed a constrained node to interact with a higher level remote DM_STORE_SRV service implementing the CIM model in a HBase store. The defines used in the HPP header files were easy to generate from the IPSO documents by substituting "_" for " ", but it was not possible to do easily in all cases due to the inconsistent use of certain characters or mixed capitalization in the IPSO documents, e.g. use of "/" in On/Off (5850), Off-Time(5853) and Minimum Off-time(5525).

6.7 Memory Use

The memory usage is given for examples of the two data models implemented. It was necessary to remove parts of the erbium-CoAP code to create space for the HPP code to run on a Tmote Sky, although parts of the Erbium and CoAP stack were retained in the runtime to allow use of the CoAP transport and the Copper Browser plugin for testing. This could be removed to reduce the memory footprint and allow more holistic architecture functionality to be included.

	Original Erbium REST Code			**Erbium + HPP Code**		
	Code (%)	**Data(%)**	**Total (%)**	**Code (%)**	**Data(%)**	**Total (%)**
Libc	8	0	7	9	0	8
Core	9	3	8	7	2	6
Network	50	74	53	50	63	52
Platform	12	3	10	10	4	9
Coap	17	17	17	11	12	11
Rest	5	3	5	2	4	2
Hpp	n/a	n/a	n/a	11	15	12

Table 1 Memory Usage of CIM Implementation

Table 1 shows the percentages of the available TMote memory (10K RAM, 48K Flash) in order to compare the memory use of key components by the original

Erbium REST er-rest-example application code and the modified code with HPP. Percentages are shown as the original Erbium code had to have some functionality removed to allow the HPP code to fit in memory. The code includes the implementation of the layers of the holistic architecture (without the DHT) and instances of the led, Service, Node, CIM_AlarmDevice and CIM_NumericSensor. The CIM instances took advantage of the object space's ability to only store those attributes from the CIM object that are available on the device to reduce memory footprint. As can be seen, the REST engine and CoAP use a small amount of memory compared to networking, which is equivalent to that for the platform and core. It can also be seen that the code and data usage of the holistic architecture is equivalent to that of CoAP, so that it is feasible to run the holistic architecture with those reduced CIM objects on a constrained device.

Code Component	Code	Data
LWM2M and OMA	7742	3807
REST and CoAP	8278	2228
Data Model and Local Instrumentation	1686	1316
Object Space (and supporting utils)	1036	139
IPSO extensions (to integrate with Data Model and Local Instrumentation layers)	4126	1684
Full Stack	60474	21753

Table 2 Memory Usage of IPSO Implementation

Table 2 shows the memory use in bytes of selected components on a WiSMote WSN node running Contiki, with instances of the IPSO LightControl, Generic Sensor and DigitalInput objects implemented on that device (and returned using CoAP as a transport). It shows the memory use of the holistic architecture's components used to store sensor and node data, i.e. the data model service, object space and local instrumentation layers. It also shows the memory use of the

existing IPSO, REST and CoAP code. This shows that the size of the holistic architecture's layers for storing data (such as from the IPSO data model) is suitable for constrained nodes, as its memory use is comparable to that for LWM2M and OMA and makes up approximately 11% of the overall code size and 14% of the overall data size, where use of both memory types is dominated by the networking code. The code in this case used dynamic memory allocation for objects, channels and endpoints, so this is not shown in this case.

	HPP Only (Text/Data)	**HPP +IPSO/LWM2M (Text/Data)**
HPP Libraries	16961/5363	16961/5363
Service layer	2828/204	2828/204
DM layer	2143/214	2143/214
Object layer	3055/1261	3055/1261
Li layer	2570/828	2570/828
DHT	3387/16	3387/16
uIP Stack	26361/4765	26361/4765
RPL	10865/250	10865/250
CoAP	-	9289/761
LWM2M	-	9641/543
IPSO	-	2671/275
TOTAL	107240/23917	130727/26291

Table 3 Memory Use of Nodes of Different Capability

Table 3 shows a more complete picture of the memory (bytes) used on a WiSMote WSN node than in Table 2. The first column shows the memory used by a node using only HPP and the second column shows a node that also included the OMA Lightweight Machine to Machine (LWM2M) and CoAP engines, with both nodes storing local LightControl and Temperature Sensor IPSO instances. In both cases,

the overall memory use is larger than in Table 2 for a number of reasons. The main reason is that the code used in Table 3 uses statically defined structures rather than dynamic memory allocation. This static memory allocation was for connections (8), channels (8 of 184 bytes each), messages including a buffer (1 per channel of 256 bytes), objects (20 of 56 bytes each), the peer (8 of 204 bytes each including the DHT identifier and IP address) and bucket objects for the DHT. The memory use is also increased by the use of RPL and the inclusion of the code for DHT (which is not included in Table 2) and some corresponding changes to the IPSO and LWM2M code. These memory sizes show that the holistic architecture meets the requirement to run on resource constrained nodes such as WiSMote, with memory use equivalent to CoAP and LWM2M. It can also be seen that the limits used in the tests of 20 objects, 6 channels and 6 peers could be increased by removing the RPL and OMA LWM2M code and freeing up about 20KB for code and data.

It is also worth noting that the memory use shown above includes code to support the HPP messages and the layers in the architecture code, as well as the test code to send specific messages (as described in section 6.8.1). This memory size could be reduced by building images for specific roles, e.g. a peer with a DM_SOURCE_SRV role only would not need code to handle *Add* or *Take* or *Notify* and a peer with a DM_FORWARD_SRV only would not need the local instrumentation layer.

6.8 HPP Performance

6.8.1 Node Functionality for Testing

A number of nodes acting in different test modes were used during the tests on Contiki and Linux. These tests evaluated the functionality of the defined messages in HPP and of the layers in the holistic architecture, but also tested the scalability and robustness of HPP and the holistic architecture. In these tests, each non-bootstrap peer begins by sending a *Hello* to a bootstrap peer to get an identifier and join the overlay. In these tests, the bootstrap peers allocated new peer identifiers randomly in order to test the bucket management, i.e. whether a new peer is in the correct range for a bucket or if it should generate a bucket split.

The test nodes were as follows (and all added their Node instance before the steps shown), with TEST_ADD_PERIOD set to 20s and TEST_GET_PERIOD to 10s:

1. A bootstrap node that acts as a peer with DM_BOOTSTRAP_SRV, DM_FORWARDER_SRV and DM_STORE_SRV roles.
2. A source test node that acts as a peer with a DM_SOURCE_SRV role. It sends the following messages to a DM_STORE_SRV peer, usually the bootstrap, which stores objects added to it. Note this node is a source of data, but not a DM_STORE_SRV:
    a. an *Add* message for a new class named "testobj" in a namespace "NS1". The object handle from the reply is stored.
    b. an *Add* message for a new class named "testobj" in a different namespace "NS2". This will re-use the object class to store the template using attributes, but create a link to a new namespace. The object handle in the reply is stored.
    c. an *Add* message for a new instance named "testinst" in namespace "NS2". The object handle from the reply is stored.
    d. A *Get* message with "NS1" "testobj" handle from the *Add* reply.
    e. The source node then enters a refresh period (TEST_ADD_PERIOD), where it periodically sends an *Add* message for the "testobj", "testinst" and its own "Node" class and instance to the DM_STORE_SRV in order to renew the lease on these objects. It was verified (with a later *Get* message) that the added "testobj", "testinst" and "Node" objects were removed from the DM_STORE_SRV peer after the lease expired when the test node was stopped.

    The leases for the "testobj" and "testinst" object were set to greater than 10 minutes (effectively infinite in these tests), but were set to 5 minutes for the "Node" class and only 10 seconds for the Node instances, so that the Node instance would expire if an *Add* was not sent in time to renew the lease.
3. A sink test node that acts as a peer with a DM_SINK_SRV role. It sends the following messages to a DM_STORE_SRV peer:
    a. A *Get* message for the "testobj" class in "NS1". Note that this will not return an object, unless there is a source test node that adds it.

b. A *Get* message using the values for the key attributes of "testobj" in "NS1" to select that object to retrieve. Note that this will not return an object, unless there is a source test node that adds it.

c. A *Get* for "Peer" object on the DM_STORE_SRV peer, where the reply holds the address and DHT identifiers of peers known to that peer. It was verified that the object from this reply is cached locally (and can be used to reply to requests from other peers), as this sink test peer also has the DM_STORE_SRV role.

d. For the tests on Linux based nodes, the sink node sent the following messages

   i. a *Get* for the "Node" class.

   ii. a *Get* for the "Node" instance with the name key attribute set to this node's name (match type set to EXACT).

   iii. A *Get* for the "Node" instance with the name key attribute set to a pre-set name and a *Get* for "Node" instance with any name key (match type set to WILDCARD).

   iv. a "Get" for the "Peer" instance with the key set to the identifier of the sending peer (and it was checked that the reply contained up to the limit of closest nodes in the reply. This limit had been set to 8 for Contiki and was set to 32 for Linux).

   The first three messages were sent in turn at the end of the first three periods (TEST_GET_PERIOD), after which the *Get* message for the "Peer" was sent every period.

e. On Contiki, the sink node then enters a refresh period, where it periodically (TEST_ADD_PERIOD) sends an *Add* message for its own "Node" class and instance to the DM_STORE_SRV and *Get* messages to get the "Peer" object from that DM_STORE_SRV.

4. A full test node that acts as a peer with DM_SINK_SRV, DM_SOURCE and DM_STORE_SRV roles.

   a. It tests source functionality by sending an *Add* message for a "testobj" class (including the attribute descriptions) and an *Add* message for a "testobj" instance. It does this periodically to ensure the leases are renewed.

b. It tests sink functionality by sending
    i. a *Get* message for a "testobj" class and a *Get* message for a "testinst" by specifying the object handles.
    ii. a *Get* "testinst" by specifying the key attributes.
    iii. a *Get* Peer by peer identifier.

5. A direct test node that acts as a peer with DM_SINK_SRV, DM_SOURCE_SRV and DM_STORE_SRV roles. This peer connects to a known peer (its bootstrap peer) and sends *Get* "Peer" messages to other peers by setting the destinationId for a node. Each peer in the path to that destination node uses HPP and the DHT to forward the message as shown in Figure 25. The *Get* "Peer" object message is sent periodically (every 10th refresh period or every 10th time after it has received a message). The peer requested from the destination is either a random peer identifier from the bucket containing the sending peer's identifier or (every 3rd time) the sending peer's identifier.

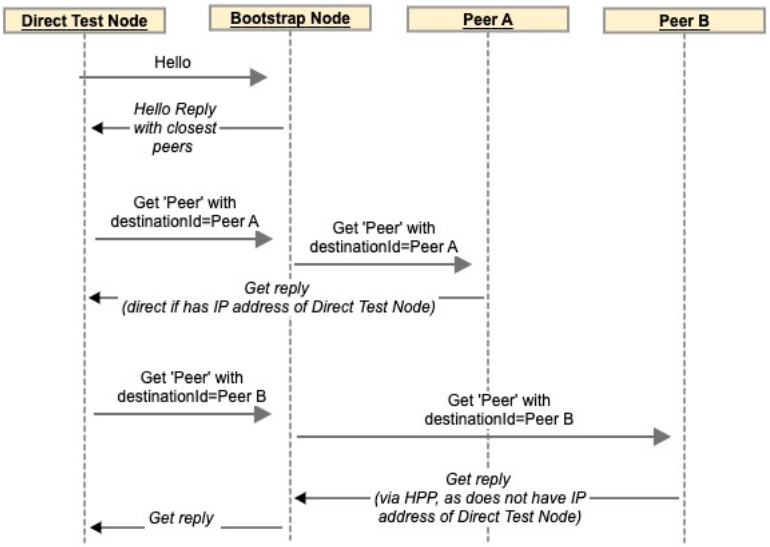

Figure 25 Sample of Test Messages for Direct Test Node

6.8.2 Constrained Device (Contiki) Tests

Several test scenarios were run on the Cooja Contiki emulator, with each being run up to 10 times. The test scenarios used the example networks shown in Figure 26 and Figure 27. These scenarios used an edge router running RPL, allowing a

CoAP or LWM2M server or external HPP Peer to access WSN nodes. The HPP nodes are located in a simple overlay network and are given the address of a bootstrap peer to send a *Hello* message to.

In these scenarios, the DHT was used to hold peer IP addresses and DHT identifiers and RPL was used to route messages in the WSN. HPP is not tied to the use of RPL, as it is designed to run over an overlay network, where the underlying networks may comprise of some nodes not running HPP and may involve different network technologies, i.e. nodes not running HPP will use IP to route HPP messages to the IP address of the peer.

The scenarios are shown for 5, 7 and 9 nodes in Figure 26 and Figure 27. The roles in the tests are set based on the identifier of the node; node 1 is the edge router; nodes 3 and 7 run a full series of tests for DM_SINK_SRV and DM_SOURCE_SRV peers; nodes 4 and 8 are DM_SOURCE_SRV peers; nodes 5 and 9 are DM_SINK_SRV peers. Node 2 acts as a DM_BOOTSTRAP_SRV peer and DM_STORE_SRV, as does node 6 in the 9 node test.

The 5 nodes in Figure 26 are set up so that nodes are all in range of the bootstrap peer service on node 2. The test messages are sent to the DM_BOOTSTRAP_SRV peer running on node 2, which also has a DM_STORE_SRV. In the 7 node test in Figure 27, nodes 6 and 7 need an intermediate node to be routed to node 2, which is still their DM_BOOTSTRAP_SRV peer. The nodes in the 9 node test in Figure 27 are all within 1 hop of their DM_BOOTSTRAP_SRV peer, either node 2 or node 6.

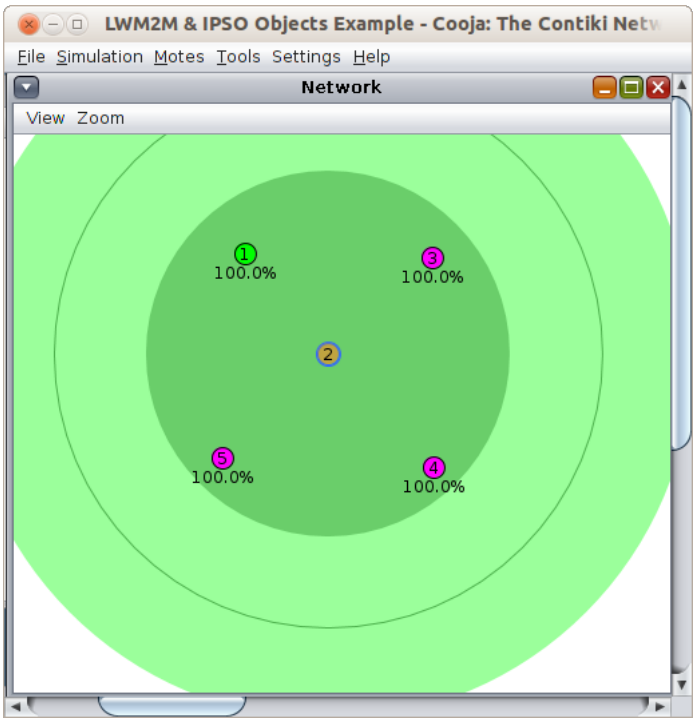

Figure 26 Simulation with 5 nodes

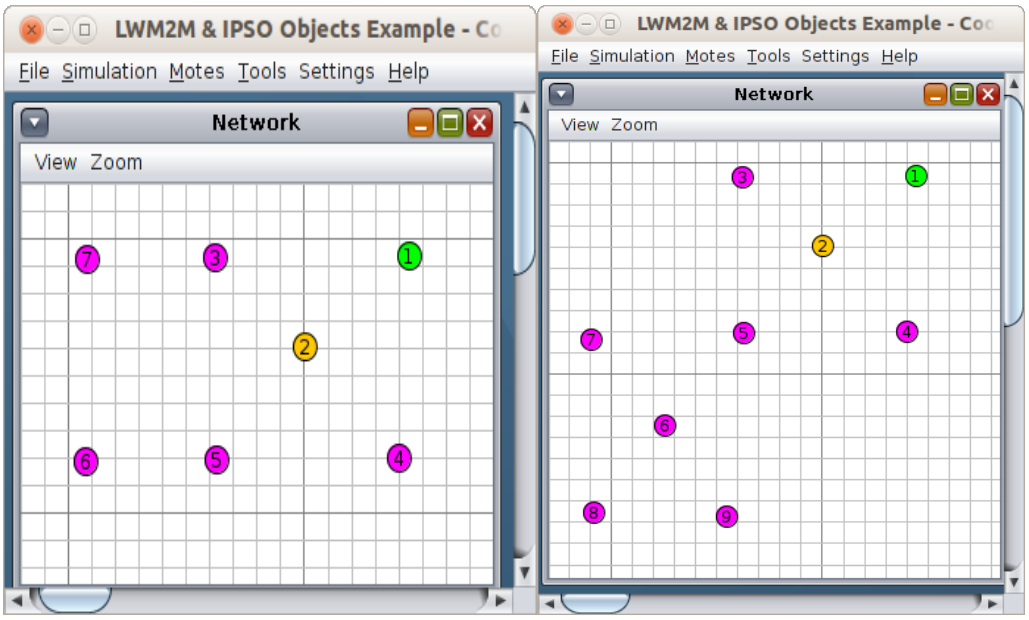

Figure 27 Simulations with 7 nodes and 9 nodes

HPP peers could, however, also be used to route messages to other peers over a number of hops, using the peer identifier to select a next hop peer from those stored in the bucket, possibly in conjunction with some stored information, e.g. previous response times.

The source and sink nodes in these scenarios ran a series of tests as described in section 6.8.1 for the source test node and sink test node.

Stack use was monitored while running the tests to ensure it did not exceed the allocated 1KB (the largest size seen was 750bytes).

	Hello Processing	**Get Processing**	**Add Processing**	**Hello Reply**	**Get Reply**	**Add Reply**
Average (ms)	31	7.8	20.8	166	208	173.8
Range (ticks)	3-5	1	2-3	20-22	22-30	18-25

Table 4 HPP Message Processing Times for 5 and 9 node tests (except node 5)

Table 4 shows the times that were stored during the tests above for *Hello* to a bootstrap peer and the time for *Get* and *Add* of objects to the node acting as a DM_STORE_SRV. The processing time shown in Table 4 is the time to process the messages on the receiving peer and to send the reply, having carried out the required store or retrieve operation. The reply times are the time from the node sending the message to when the reply is received on that node a single hop away (to focus on processing time rather than routing). The results were consistent in the 5 and 9 node tests for all nodes, except node 5 which had an average reply time of 380ms, due to the processing of RPL messages to nodes 6, 7, 8, 9 overlapping with HPP messages. It is worth noting that node 3 had about 1 in 10 replies outside the range shown (and up to 3 seconds) in the 9 node test. Also, times were similar for nodes 3,4,5 in the 7 node scenario, but reply times for nodes 6 and 7 had an average of 365ms.

The message processing times on the receiving peer in Table 4 did not depend on the number of objects being searched or added, albeit there was at most 20 objects (due to the static memory allocation of the related structures). The reply times showed more variability and the times for *Hello* and *Get* replies are larger than for *Add.* This is because the *Add* reply is smaller in size and the *Get* reply may require more than one message fragment depending on the size of the object retrieved. These results suggest that HPP is feasible on constrained nodes, even when using

a string format, rather than the binary encoding of HPP that could be implemented.

The robustness of HPP was demonstrated during the Contiki tests above by stopping nodes during those tests and observing that their peer information and any added objects were deleted on the DM_STORE_SRV and other nodes, when their leases expired. Restarting the node that was stopped and running the tests on it again resulted in its peer information and objects being added by HPP to the node with the DM_STORE_SRV role.

6.8.3 Linux Node Tests

The Linux implementation was used to test the scalability and robustness of the holistic architecture and HPP at a larger scale, as this could be done more easily than on Contiki. The tests were run on a laptop running Ubuntu 18.04LTS with a 2.3GHz Intel i3 processor and 4GB of RAM.

6.8.3.1 Scalability Tests

In terms of scalability, it is required to ensure that the potential of P2P to add peers easily is not limited as the number of peers increases. The scalability of P2P has been shown by the scale achieved by BitTorrent and its use of Kademlia. For this reason, the goal of the tests in this section is primarily to show that this scalability and robustness are still present in the incorporation of the Kademlia-based DHT into HPP, rather than an absolute test of the limits of scalability or of optimising performance in terms of response times.

Furthermore, it is worth recalling that the DHT is used in HPP to identify peers and not to place data, as the peer is the source of sensor data, unlike in other P2P use cases where data is stored based on the DHT identifier. Hence, the tests in this section focus on finding identifiers using the *Hello* message, retrieving data using the *Get* message and sharing data between peers using the *Add* message, where the destination is determined by a peer and not using a placement approach based on an identifier.

These tests used nodes in the scenarios shown in Figure 28 and Figure 29, which are intended to be representative of certain WSN use cases. For example, Figure

28 shows a bootstrap node with other nodes in a star topology, as would be the case in IEEE 802.15.4 with Reduced Function Devices (RFDs) connecting to a Full Function Device (FFD) at the centre of the star. In a WSN, the DM_BOOTSTRAP_SRV peer running on a node in each figure could also act as the border router as shown in the preceding Contiki tests. The tests send *Get* messages which use the TCP address of the node (derived from an entry for its identifier in the DHT) expected to have the data, except for the direct test node which uses HPP forwarding with a destination identifier specified in the message. The topologies shown in Figure 28 and Figure 29 also allow data to be cached by the bootstrap node (as it also has a DM_STORE_SRV role) when it handles replies to *Get* requests using HPP forwarding (from the direct test nodes), as described in section 4.5.5.4.

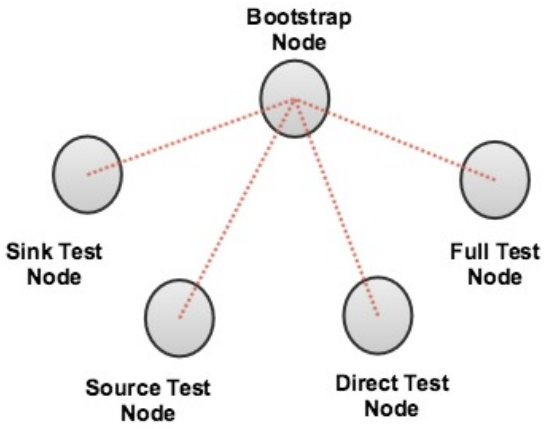

Figure 28 Single Bootstrap Node Test Scenario

The tests based around the topology shown in Figure 28 tested the scalability of increasing numbers of peers, in the roles described in section 6.8.1, connected to a single bootstrap peer as follows:

- "1bs-sink-test" used 4, 20, 50, 100 sink test peers to the bootstrap peer and is focused on *Get* messages.
- "1bs-source-test" used 4, 20, 50, 100 source test peers to the bootstrap peer, including *Add* messages tested with corresponding *Get* messages.

- "1bs-mixed-test" used an equal mix of sink, source test peers (with 1 extra sink if necessary to make the total) and one full test peer in totals of 4, 10, 20, 50, 100 peers to the bootstrap peer.
- "1bs-direct-test" used an equal mix of sink, source and direct test peers and one full test peer in totals of 4, 50, 100 peers to the bootstrap peer

Figure 29 shows the "multi-bs-test" scenario, where bootstrap peers connect to each other in a peer-to-peer manner. This allows the node identifiers to be known across each of the sub-networks of peers so that peers can form an overlay, which allows HPP forwarding to be tested. This tests the scalability when peers (in this case the bootstrap peer) are connected to more than one other peer as follows:

- 10 test peers (1full, 3 source, 3 sink and 3 direct) to 1 bootstrap peer,
- 50 test peers (1full, 3 source, 3 sink and 3 direct) to each of 5 connected bootstrap peers
- 100 test peers (1full, 3 source, 3 sink and 3 direct) to each of 10 connected bootstrap peers

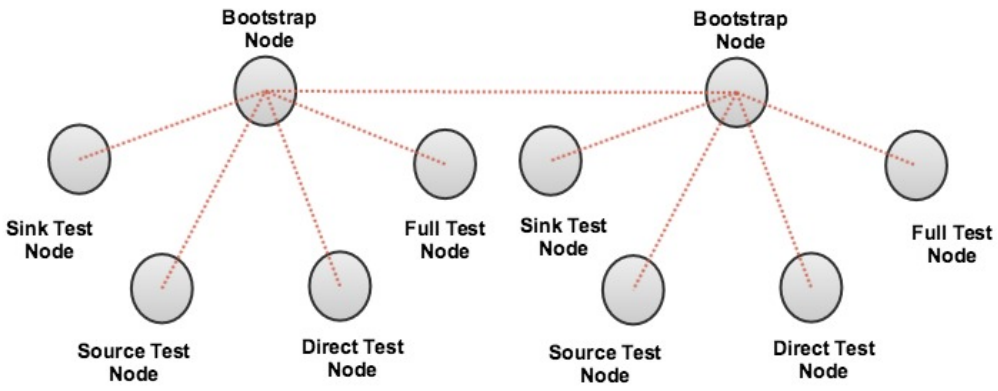

Figure 29 Multiple Bootstrap Peer Test Scenario

The following tables and figures showing results are based on the following:

- The times shown for the receiving peer are the time to process the request and send the reply.
- The times shown for replies are from when the request was sent to when the reply was received and processed on that requesting peer, e.g. the time

for a reply to a *Get* "Peer" will include the time to add any new nodes in the reply to the appropriate bucket on the requesting peer.

- The tests were run up to 3 times and the results were comparable for each run. The result times and numbers of messages in the tables and figures are presented for a particular test run, so that the various counts and times are comparable. This also shows the occasional outlier time or where *Get* "Peer" or *Hello* messages were sent as a result of retries or refreshes.

The maximum number of peers in a reply to a *Hello* and *Get* "Peer" was set at 32. This is more likely to be set to the value of k (usually 8 as used in the Contiki tests) in a real deployment, but this value was used in order to test further the robustness and scalability of the system, e.g. it involved reading from multiple buckets. The value of k is governed by the number of simultaneous failures in the refresh period as per section 3.7.3.5. This maximum value of 32 meant that the size of a reply message containing the peer identifier and IP address of 32 peers was up to 1368 bytes for just the peer identifiers and addresses.

Initial tests showed that debug and logging messages containing the identifier and address information for large numbers of peers increased the times to send replies significantly as the number of peers increased. The logging was reduced as a result of this, but given that these tests were intended to demonstrate scalability, rather than providing the best run times, some debug and logging messages were retained in order to verify correct operation of the tests and gather results. It was ensured, however, that the content of these messages was limited as the number of peers grew. During each test run, the results were recorded into a named log file using periodic log messages containing comma separated fields holding the data from each node, i.e. the number and times of the different message types sent, the number of replies and the count of certain bucket operations. These logs were then parsed as csv files to get the results shown in this section.

The tests sent a number of *Get* or *Add* messages at the start of the test and then entered a mode of sending certain messages periodically as described in section 6.8.1. All the tests started the bootstrap peer and waited 10 seconds to ensure it had initialised and then started the non-bootstrap peers 2 seconds apart. The non-bootstrap peers started running their functional tests (as per section 6.8.1),

resulting in the processing of different types of messages in different sequences depending on the start-up of peers, particularly at the bootstrap peer.

During these tests, the peers sending requests wait for a reply for each request before proceeding to send the next request, meaning that a slow reply extends the time taken on a peer and adds variability to the results. This could result in a reply that indicated a failure of a *Get* (as the object is not yet added to that node) or could result in a retry for a *Hello* request as a result of a slow reply. Once all the specified peers had started, the test ran for 5 minutes before each peer was killed, in order to test the refresh logic and the periodic sending of messages. This peer behaviour is somewhat representative of a real scenario and acceptable as the goal was to demonstrate the handling of multiple messages, rather than the best time per message or throughput per node.

A peer refresh period of 1 minute was used in order to accelerate the running of tests by testing the refresh conditions more in a given time and loading the system more than would be expected in a real deployment. Longer refresh periods are more likely in real deployments, depending on how often nodes join/leave or fail, e.g. as per section 3.7.4 BitTorrent refreshes peers that it has not received a message from in the last 15 minutes.

Figure 30, Figure 31 and Figure 32 show the server processing and reply times. These times were reasonably consistent across a range of tests with varying message types, peer roles, number of messages and the number of nodes, although an occasional very long reply time was seen, e.g. in the 1bs-direct-test in Figure 32. The following observations can be made on the times for requests and replies:

- The number of requests handled by the bootstrap peers can be seen in Table 6 according to the test and number of peers. The corresponding processing times in Figure 30 for these tests indicate that the number of requests did not necessarily affect the processing times.
- Figure 31 shows that the times across the different bootstrap peers were reasonably consistent in the 50 node and 100 node “multi-bs-test”.
- Figure 30 and Figure 31 show the *Get* time across the tests is affected by the mixture of *Get* requests and replies sent. Specifically, as the number of nodes increases, the number of peers in *Get* “Peer” replies increases and it

takes longer to process. For example, the 1bs-source-test processed the same number of "Get Peer" messages as the 1bs-sink-test, but only a few *Get* messages (at the start) and Table 5 shows that its average and median *Get* times are higher, which is due to the time for the larger *Get* "Peer" messages. Consequently, the maximum time to process on the receiving peer tends to increase. Figure 30 does, however, also show a decrease in the times for *Get* in the 100 node tests compared to 20 nodes. This is due to the number of *Get* messages which return quickly with a short message when no object is found, e.g. when the Node object was expired in the cache as its lease had not been renewed by an *Add* (this is more likely as the number of nodes increased as the requesting peers waited for a reply before sending a message). For example, in the "1bs-source test" no *Get* messages are processed in under 100ms for 20 nodes for an average of 186ms, whereas for 100 nodes there were 34 out of 616 processed in under 100ms for an average of 167ms (which is 174ms excluding those less than 100ms). This effect can also be seen in the minimum value of the *Get* time for the 1bs-mixed-test for 20 nodes shown in Table 5.

- As seen in the Contiki test results in section 6.8.2, the times shown in Figure 30 and Figure 31 indicate that the message processing times on the receiving peer (usually the bootstrap peer) do not depend on the number of objects being searched or added, as the number of peers increased. Conversely, as explained above the number of *Get* messages that do not return an object can reduce the *Get* time.
- Similar to the results seen in the Contiki tests in section 6.8.2, Figure 30 shows that an *Add* takes longer than a *Get* to process on the receiving peer, but Figure 32 shows that the reply is received more quickly on the requesting peer and the *Get* reply times showed more variability. This is because the *Add* reply is smaller in size, as the *Get* reply will contain an object if successful.
- Figure 30 and Figure 31 show that there was some variability in the times to process *Hello* messages on the bootstrap peer as the number of nodes varied. The times did not necessarily simply follow the increase in nodes and this would seem to be due to the number and mixture of messages processed, with the median and average times being similar in each of the 1bs-sink-test and 1bs-source-test.

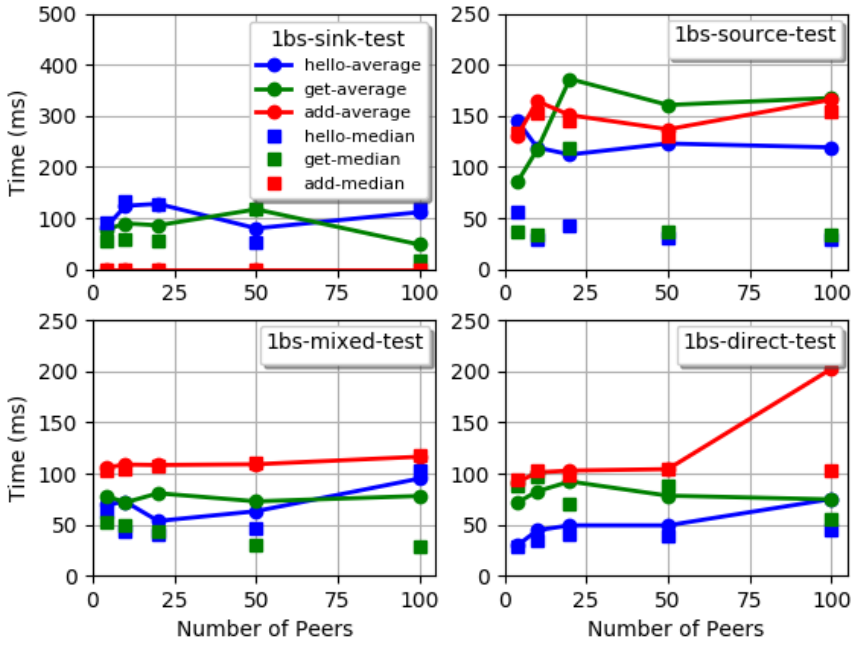

Figure 30 Request Processing Times on Bootstrap Peers

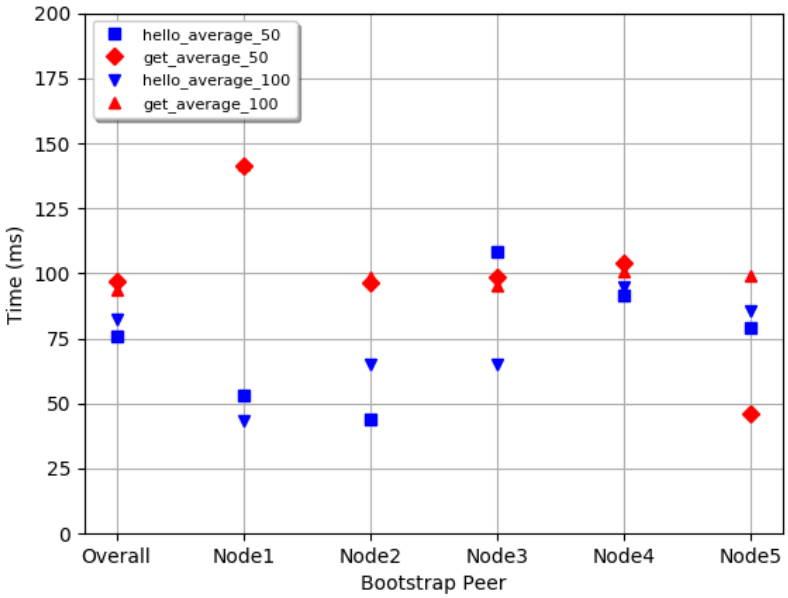

Figure 31 Request Processing times for Multiple (50 or 100) Bootstrap Peers

Note that get_average_100 and get_average_50 on Node1 and Node2 are coincident in Figure 31

The reply times in Figure 32 at the requesting peer showed much more variability than the times to process the message on the receiving peer, due to the time spent building, sending and receiving larger reply messages. The time to process messages on the receiving peer was affected by the mixture and number of messages processed as the number of nodes, and hence requests, increases. This can be seen in that the maximum times and the average times are generally higher, with occasional very high values as the number of nodes increases. The difference in request processing and reply times with the greater range in reply times is shown in Table 5 for 20 nodes in the "1bs-sink-test" and "1bs-source-test". The standard deviation did tend to increase as the number of nodes increased due to this increase in nodes making a larger reply more likely and causing a larger reply time, but the median did not increase unduly as the number of nodes increased indicating the effect of the less frequent longer replies.

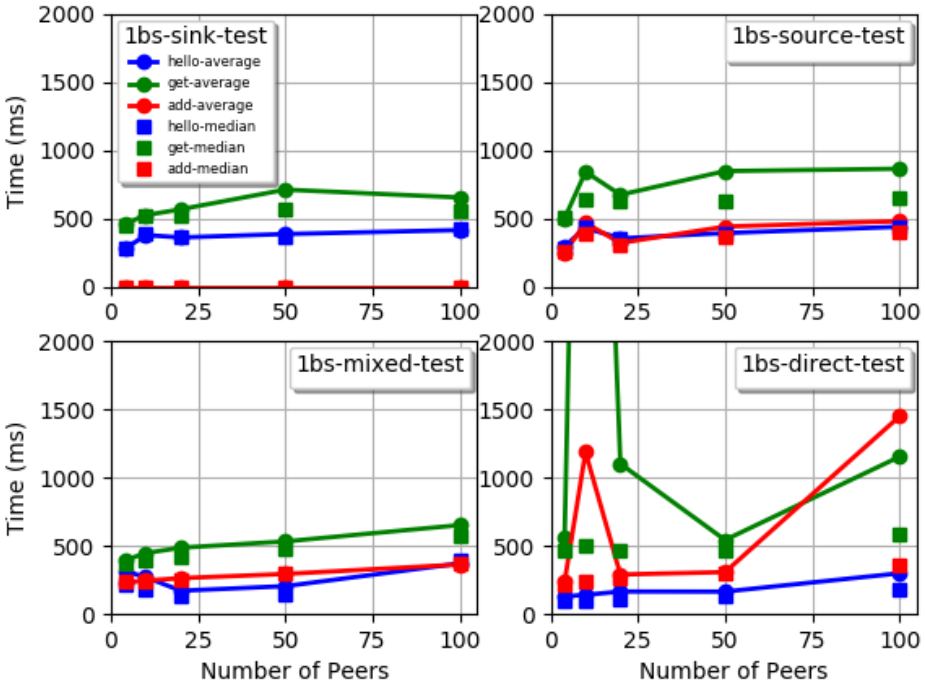

Figure 32 Times to Receive Replies on Non-bootstrap Peers

Test	Role	Msg Count	Avg	Median	Min	Max	StdDev
1bs-sink-test	Bootstrap	334	85.68	56.5	25	261	51.52
1bs-sink-test	Non-BS	334	566.41	514	287	1421	196.98
1bs-mixed-test	Bootstrap	211	80.7	43	4	359	69.62
1bs-mixed-test	Non-BS	211	487.26	420	54	2386	262.33
1bs-source-test	Bootstrap	100	186.07	180.5	119	354	46.17
1bs-source-test	Non-BS	100	672.63	621	384	1214	189.08

Table 5 Variability of Get Message Times (ms) in 20 Node tests

Table 6 shows the number of requests handled on the bootstrap peer during the tests used for the times in Figure 30, Figure 31 and Figure 32. Note that the total for *Get* includes the *Get* "Peer" messages.

For non-bootstrap peers, the number of *Get* replies at nodes not acting as DM_STORE_SRV ranged from 1 to 15 and *Get* "Peer" replies ranged from 1 to 11 across all the tests depending on the test topology and number of peers. Replies to Add "Peer" ranged from 0 (sink peer tests) to 35 in the 50 and 100 peer test, as these tests took longer to complete owing to starting peers in sequence.

Test	**Number of Nodes**	***Hello* Messages**	***Add* Messages**	***Get* Messages**	***Get* "Peer" Messages**
1bs-sink-test	4	4	0	99	20
	10	10	0	253	50
	20	20	0	521	100
	50	50	0	1421	270
	100	100	0	3241	615
1bs-source-test	4	4	135	20	20
	10	10	339	49	44
	20	20	704	100	100
	50	50	1931	269	226
	100	100	4531	620	616
1bs-mixed-test	4	4	70	51	21
	10	10	172	139	50
	20	20	342	293	101
	50	50	849	762	245
	100	100	1948	1680	543
1bs-direct-test	4	4	36	50	20
	10	10	92	115	45
	20	19 retry started	206	299	182
	50	50	104	164	75
	100	100	215	286	120

Table 6 Number of Requests handled by Bootstrap Peers

Table 7 shows the values for following counters on the bootstrap peer during the tests:

- "num-peers" refers to the number of peers that are in the peer's buckets

- "num-finds" refers to *Get* "Peer" requests made to a peer, which is in the range of a bucket on the peer, where no peer in that bucket has received a reply within the refresh period (set to 60 seconds).
- "dropped-adds" refers to the number of times a peer is not added to a bucket due to the management of full buckets.
- "num-splits" refers to the number of times a bucket was split in order to accommodate identifiers within range.
- "cached-pings" refers to Get Peer requests that were generated at refresh time that used a peer that had been added to the cache and not to a full bucket.

These counters were also used to verify the tests, e.g. the number of requests received by all sink and source peers was 0, the number of *Add* replies received by a sink peer was 0 (as it never sent an *Add*), the number of *Hello* replies at non-bootstrap peers was 1 (unless it had retried as a result of a timeout). Also, during some of the 50 and 100 node tests, a *Get* count of 1 was occasionally seen on a peer, indicating the bootstrap peer was sending a Get "Peer" as it had not heard from this peer. Similarly, on the Bootstrap peers, counts for replies were all 0 as this peer did not initiate requests, except for a *Hello* reply from another bootstrap in the multi-bs-test and the replies to the *Get* "Peer" requests from the direct test peer in the 1bs-test-direct test.

Table 7 also shows that the "num-peers" for the bootstrap peer in the multi-bs-test was larger than the number of directly connected peers due to the sharing of peer information from the other bootstrap peers and it was also noticed that this increased gradually from the initial numbers seen in *Hello* replies as peer information was refreshed.

It can also be said that the DHT operated successfully in the tests, as shown by the following points from Table 7 and also that there was no storm of requests due to refresh or find as the number of nodes increased:

- the number of peers known about at both bootstrap peers and non-bootstrap peers grew according to the test.
- as the "num_peers" increased, the number of buckets and identifiers grew per the bucket management policy. It was seen that "num-adds" equals

"num-peers" plus the "dropped-adds", so that the amount of stored peer information does not grow indefinitely as more peers are discovered. This bucket management policy could be changed to allow all peers to be stored.

- "num_splits" counted the number of times a full bucket was split and it showed that buckets were split as more peers were added and aligned with the number of peers held in the buckets on a peer.
- the *Get* "Peer" messages (included in the Get message count in Table 6) were sent on a refresh period, every 60 seconds, so that during the 5 minutes after the first peer was started, there could be a *Get* "Peer" every minute from each peer. This was, however, reduced if the peer had received a reply from a known peer or if it was slower completing its initial tests (due to its replies being slower due to a large number of nodes sending to the bootstrap peer). Hence, it can be said that the number of *Get* "Peer" messages as a result of refresh/find did increase as the number of nodes increased, but it did so at a reasonable rate.
- "cached_pings" are when a Get "Peer" (PING in Kademlia) was sent as a result of a peer being put into the cache when a bucket is full. They are only seen as the number of nodes increase and the likelihood of trying to add a new peer to a full bucket increases (the num_finds is 0 on the bootstrap peer in the tests). These cached pings are the result of storing objects when a bucket is full and used the lease in the object space, allowing a node flexibility in adjusting leases to manage its storage.
- Columns in Table 7 indicated with a * were those where the hello count indicated that the *Hello* message had been retried from peers.

Test	Num.of Nodes	num_ splits	num_ dropped	num_peers	num_buckets	cached_pings
1bs-test-sink	4	0	0	4	1	0
	10	1	0	10	2	0
	20	2	2	18	3	0
	50	3	19	31	4	1
	100	5	62	38	6	2
1bs-test-source	4	0		4	1	0
	10	1	0	10	2	0
	20	1	4	16	2	0
	50	4	18	32	5	3
	100	5	62	38	6	3
1bs-test-mixed	4	0	0	4	1	0
	10	1	0	10	2	0
	20	1	4	16	2	0
	50	4	22	28	5	2
	100	4	62	38	5	3
1bs-test-direct	4	0	0	4	1	
	10	1	0	10	2	0
	20	2	1	18	3	0
	50	4	19	31	5	0
	100	5	58	39*	6	0
multi-bs-test	10	1	0	10	2	0
	20	1	0	8-11*	2	0
	50	1	0	9-11	2	0
	100	1	0	9-11	2	0

Table 7 Counts for Bootstrap Peers

Table 8 shows that num_finds was quite low on the sink, source and direct test nodes relative to the number of peers known about (as shown by num_peers) and it was seen to be 0 on the bootstrap servers (apart from an occasional 1 in some tests for 100 nodes), even as the number of nodes increased. This indicates that sending *Get* and *Add* messages and processing the replies kept the stored peer information updated in many cases and so avoided an explicit Get “Peer” as a find, except when the reply was not timely enough (and increasing the refresh period would have reduced the count further).

Test	Num. of Nodes Started	num_peers	num_buckets	num_finds
1bs-test-sink	4	3-4	1	3-5
	10	5-6	1	4-5
	20	7-12	1-2	4-5
	50	9-16	1-2	5-6
	100	6-21	1-2	5-8
1bs-test-source	4	3-4	1	4-5
	10	7-10	1-2	4-5
	20	7-8	1	5-6
	50	9-16	1-3	5-8
	100	7-23	1-3	5-8
1bs-test-mixed	4	3-4	1	3-4
	10	1-9	1-2	4-5
	20	7-8	1	5
	50	5-6	1	5-6
	100	1-9	1-2	0-1
1bs-test-direct	4	4	1-2	1-5
	10	4-10	1-2	2-5
	20	2-9	1-2	3-5
	50	1-9	1-2	1
	100	1-7	1	0-1
multi-bs-test	10	6-8	1	0-1
	20	1-8	1	1-3
	50	1-11	1	1-3
	100	1-11	1	1-3

Table 8 Counts for Non-bootstrap Peers

In summary, the results of these tests indicate that a HPP based approach is scalable, even when using a string format for messages, although a binary encoding of HPP would be of benefit as it would reduce the message size.

6.8.3.2 Robustness (Failure Handling)

The robustness required of HPP is primarily about ensuring the correct operation of the following elements designed to provide robust behaviour when there are changes in the number of active nodes as nodes appear and disappear:

- The operation of *Hello* messages in terms of retrying the send to a bootstrap node as a node joins/leaves, so that a bootstrap does not have to

be a single point of failure. The retry was observed in a number of tests in section 6.8.3.1, where the count of *Hello* requests was greater than expected.

- The retry of failed *Hello* messages. In some test runs of the "multi-bs-test", a bootstrap peer did give up retrying to connect to its configured bootstrap peer after a set number of 10 attempts (10s apart) and terminated. The settings for timeout and retry attempts were not increased, as these settings acted as test of robustness and logs were checked to ensure that the connected peer's information was updated.
- The expiry of a lease removing objects from the cache, especially if the node fails or moves. This was observed in the tests in section 6.8.3.1 by objects expiring when their lease was not renewed in time (as occurred in some cases in the tests with more nodes). It was also observed in those tests by allowing the bootstrap to remain running when the non-bootstrap peers were killed - in some cases the TCP socket was not removed immediately and the normal object removal did not happen on the bootstrap, but the objects were cleaned up on expiry of their lease.
- The management of buckets as peers were added and as part of the periodic "find round". This was observed in the handling of the full bucket situation in the tests in section 6.8.3.1.

The first item above regarding nodes joining/leaving was not specifically tested in section 6.8.3.1, so a specific test was performed to verify the behaviour when peers are added and removed. This test adapted the "1bs-direct-test" by starting all the peers and then terminating one third of those peers by selecting from their process identifiers at random. A new sink test peer was started as each of these peers was terminated. Once this was complete the test continued for 5 minutes to ensure the refresh behaviour was as expected. It is important to note that not all peers will be added to the buckets based on the way the full bucket is handled. For this reason, a test adding/removing peers cannot simply report on achieving the full number of peers in every peer.

Table 9 gives the "num_peers" count for the test with termination. It shows that the number of peers on the bootstrap peer does not equal the number of adds attempted, due to bucket management and identifier assignment. It can be seen, however, that it does reflect the removal of the terminated peers and the addition

of new peers, e.g. for the 20 peer test, 19 were successfully added in the initial *Hello* message processing, 6 were killed and 6 were added by *Hello* of which 5 were successful giving a total of 18.

Nodes Started	**Start num_peers**	**Peers Terminated**	**Peers Added**	**End num_peers**
10	10	3	3	10
20	19	6	6	18
50	31	16	16	29

Table 9 Counts for "1bs-direct-test" with Termination (kill)

Table 10 gives the count and message times for the 20 peer tests with and without termination (other number of peers show similar times). It shows that there was an increase in the number of *Hello* messages, due to the extra peers being started after some of the initial peers were terminated, but no significant effect on the processing times for *Hello*.

Test	**Msg Count**	**Avg**	**Median**	**Min**	**Max**	**StdDev**
Without Kill	19	49.26	41	28	207	39.04
With Kill	26	58.19	43	28	196	39.25

Table 10 20 Peer "1bs-direct-test" Hello times with and without Termination

6.9 CacheL

This section considers CacheL in terms of its suitability for implementation on a WSN node based on implementing it on Contiki. The holistic architecture is also designed to be used on more powerful nodes than in the WSN, so this section also considers CacheL's effectiveness as a cache and its use of a lease by comparing the hit/miss ratio to that for LRU. It also outlines optimisations that were made, based on the analysing the counts of key variables added in the code.

6.9.1 Implementation Complexity

The code extracts in 4.2 show that CacheL is straightforward to implement, including its use of the holistic architecture's object space and data model service layer abstractions. The C implementation consisted of approximately 150 lines of C code and used about 1KB of memory with a supporting object abstraction using a circular list library. The CacheL implementation was about 200 lines of Java code whereas the LRU code was 30. This slight increase in code size is reasonable, given that the CacheL Java implementation also includes code for lease handling and interfacing to the object store, as well as the extra code due to the use of HashMap, which was used to be similar to the LRU code in that aspect. CPU use was observed and was found not to be an issue during the test runs.

6.9.2 Performance Comparison of LRU and CacheL using YCSB

YCSB was used to compare the effectiveness of CacheL to LRU using relatively large sets of data (compared to what would be on a sensor device). The YCSB driver was modelled as a DM_STORE_SRV cache with the object API methods of add(), get() and remove() integrating easily with the YCSB API methods of init(), read(), delete(), update(), insert().

Load	Ratio of Operations	Distribution	Nature	Example
WA	Read/update : 50/50	Zipf	Update Heavy	Session store
WB	Read/update : 95/5	Zipf	Read Heavy	Photo tags
WC	Read/update : 100/0	Zipf	Read Only	User Profile
WD	Read/update/insert: 95/0/5	Latest	Read Latest	Status updates
WE	Scan/insert: 95/5	Zipf	Short Ranges	Posts in thread
WF	Read/read-modify-write: 50/50	Zipf	Read-Modify-Write	User Database

Table 11 YCSB Workloads

These tests were not intended to be fully representative of real WSN scenarios, but the role of CacheL is not necessarily limited to WSN nodes in the holistic architecture. YCSB testing was performed on a single dual core PC with 8GB RAM, using the default YCSB data size of 1000, 1 KB records and 1000 operations.

These tests used the configurable YCSB workload options shown in Table 11where WA indicates Workload A. A workload with a Uniform distribution (WB-Uni) was added to the tests in order to represent periodically reporting sensors. The tests were run for LRU and CacheL (with and without leases) over a range of cache sizes. During these tests, CacheL made the fronthand/backhand sweeps based on the number of calls to add() for ease of integration with YCSB, rather than sweeps being based on the time since the last sweep, which would have suited CacheL better and is likely in a WSN scenario.

6.9.2.1 LRU vs CacheL without leases

This section considers the hit/miss ratio of CacheL compared to LRU for the case of no lease. This is not its intended use case and is included as an indication of its effectiveness as a cache and for comparison to the case where it is using leases to expire the cache contents. CacheL works like CLOCK in this case. Table 12 shows the hit ratios were comparable for LRU and CacheL, with LRU slightly higher on Workload B for Uniform distribution, but CacheL was equal or better on Zipf tests. Changing the time or number of adds (which had a default of 3) between fronthand and backhand sweeps did not affect the hit ratio, due to the short duration of the tests.

Cache Size (# of Entries)	100		250		500		750		1000	
	LRU	ChL	LRU	ChL	LRU	ChL	LRU	ChL	LRU	ChL
WA-Zipf	56.4	55.6	58.4	59.9	59.2	59.5	61.7	61.8	63.9	62.6
WB-Uni	15.6	12.9	29.5	29.1	52.6	48.7	73.5	77.0	100	100
WB-Zipf	19.5	23.1	32.0	30.8	56.7	56.6	78.7	78.8	100	100
WC-Uni	10.2	11.0	28.9	24.9	52.7	49.5	72.3	77.3	100	100
WC-Zipf	9.4	19.2	23.8	26.1	47.8	50.6	71.9	73.0	100	100
WD-Lat	69.7	62.4	83	82.2	91.5	91.8	95.6	96.7	99.4	99.9
WD-Uni	10.7	11.5	27.5	27.3	50.3	50.7	62.3	74.9	96.7	98.2
WF-Uni	9.0	9.0	28.1	25.2	51.4	52.7	63.3	73.0	100	100
WF-Zipf	19.1	20.4	35.9	36.1	44.7	58.2	68.1	79.1	100	100

Table 12 LRU and CacheL (Without Leases) Hit Ratios per Workload

6.9.2.2 LRU vs CacheL with leases

These tests use leases uniformly distributed over a time range of 0-100ms or a time range of 0-1000ms. These ranges are smaller than could be expected in a real WSN, but were used as the tests completed in approximately 300ms. These time ranges show how effectively CacheL manages leases, although not taking advantage of CacheL being able to use a 'hint' provided by the application/sensor for an appropriate setting for a lease. Comparing the results in Table 12 to Table 13, the use of CacheL with the 0-100ms lease distribution generally has a hit ratio

higher than, or comparable to, LRU for cache size 100 and is comparable at other cache sizes, although there is a notable reduction for the Workload D-latest distribution (WD-Lat). Table 13 shows the hit ratio is reduced for CacheL in all 100 sized caches when using the 1000ms lease compared to the 100ms lease range as residency in the cache is dominated by the priority given to lease. It also suggests the granting of leases should be managed actively to ensure appropriate expiry, e.g. based on the leases remaining in the cache.

Cache Size (# of Entries)	**100**		**250**		**500**		**750**		**1000**	
Lease Range	**100**	**1000**	**100**	**1000**	**100**	**1000**	**100**	**1000**	**100**	**1000**
WB-Uni	20.0	13.6	28.1	27.5	52.9	54.0	75.4	75.7	99.9	99.8
WB-Zipf	16.3	13.4	27.5	29.5	51.2	54.0	74.3	75.5	100	100
WC-Uni	17.6	11.3	25.5	25.0	50.7	52.3	76.7	74.8	100	100
WC-Zipf	15.7	9.8	22.6	26.7	48.7	48.5	73.1	75.3	100	100
WD-Lat	48.6	5.6	77.4	9.9	90.7	54.2	97.4	22.2	99.7	57.0
WD-Uni	28.3	13.7	25.8	30.0	51.0	51.0	73.6	76.7	96.4	99.7
WF-Uni	21.3	10.0	26.1	24.1	49.4	50.6	73.9	75.3	100	99.9
WF-Zipf	21.5	10.0	23.1	26.3	48.5	52.0	73.5	75.6	100	100

Table 13 CacheL (With Leases) Hit Ratios per Workload

The results show that CacheL is comparable to LRU and it expires data quickly in a small cache, when using short leases, which is what it is required to do on

constrained WSN nodes. It can also be seen that the algorithm manages leases effectively and could take advantage of a lease provided by source nodes as a hint.

6.9.3 Optimisations to CacheL

A number of optimisations were made during testing to improve the use of access counts and to reduce unnecessary sweeps of the cache queues. The first optimisation was to hold a reference to the item with the minimum lease when a sweep does not find an expired lease. This avoids an unnecessary sweep on the next backhand if the time of the minimum lease has not been reached. This improved the latencies relative to LRU, especially for the lease range 0-1000 ms, e.g. the backhand sweep was skipped over 900 times for Workload A at all sizes and for other workloads according to cache size, e.g. 944, 795 for Workload D (Uniform) for sizes 100, 250.

The second optimisation was the addition of a new method adjustLeaseByCnt(), which increased the lease by the backhand period for items with an access count above zero. This extends the time frequently accessed items stay in the cache while still giving priority to the lease and allowing smaller leases to be granted. This worked for small leases, but had little effect for long leases, e.g. cache size 100 and workloadb-uniform had 892 failed puts out of 1000 as no cached item had expired. This optimisation is expected to be useful in longer-lived tests.

The third optimisation was the addition of another sweep after the backhand if it did not delete the required number of items, as the first sweep may have moved items to the pendingQ or decremented access counts.

A small improvement to the implementation was made during the tests to hold the last_lease_check time at the end of the main sweep in order to reduce the number of system time-related calls made.

6.9.4 Performance Characteristics of CacheL

This section considers the performance characteristics of CacheL itself rather than just considering its hit/miss ratio. This was done by adding counts for the number of calls made to key methods in the code.

Figure 33 shows the values of these counts for cache size 100 and a lease distribution of 0-100ms. This cache size and lease distribution is where CacheL was most effective, i.e., expiring data quickly in a small cache. fhDeleteCount and bhDeleteCount count the number of deletes done on a fronthand or backhand sweep as leases expire. It can be seen that the fronthand sweep had an effect on the WA-ZIPF, WD-Uni and WF-ZIPF workloads, although fhDeleteCount drops to less than 10 for other cache sizes. The maximum number of entries on the pending queue, pendQMax and fhDeleteCount show that the pending Queue was used for cache size 100 and Workload A. SweepDeleteCount counts the number of items deleted after a backhand sweep and shows the impact of the third optimisation above of an additional backhand sweep on bhDeleteCount. bh_skipped counts the backhand sweeps not run based on the minimum lease in the cache, showing the value of that first optimisation above.

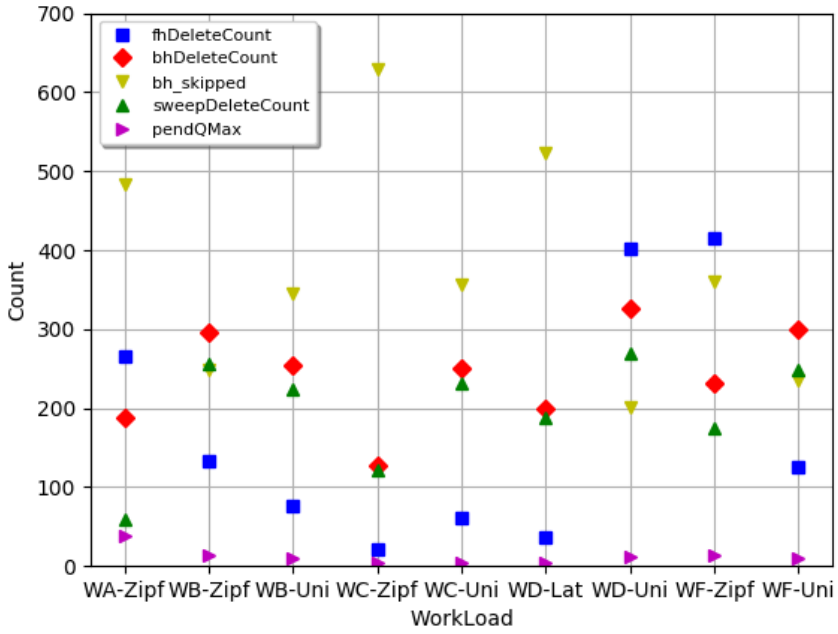

Figure 33 Counts of key variables for Cache Size 100, Lease 0-100ms

Other counters showed that when long leases were used there was limited value in using the access count in cache replacement and in using the fronthand sweep, e.g. the 0-1000ms distribution had few leases that were sufficiently close to expiry to populate the pending queue.

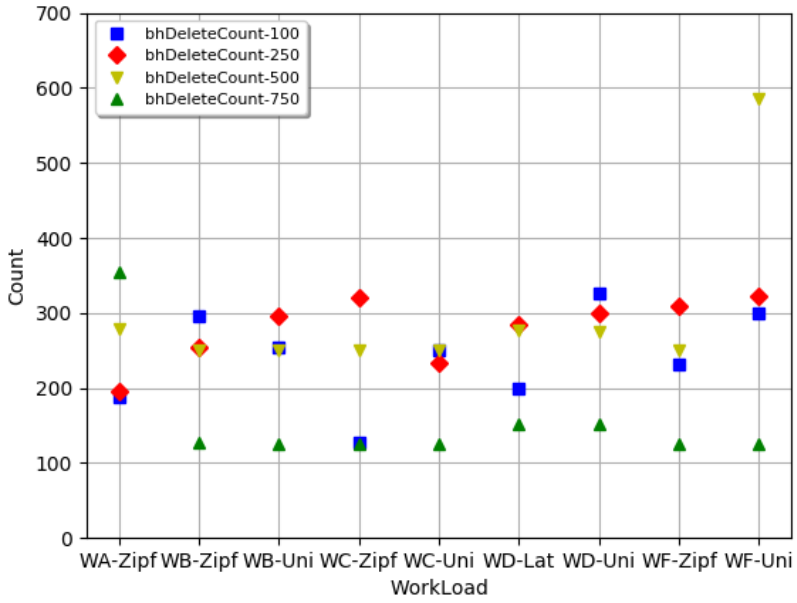

Figure 34 Backhand Delete Counts per Cache Size, Lease 0-100ms

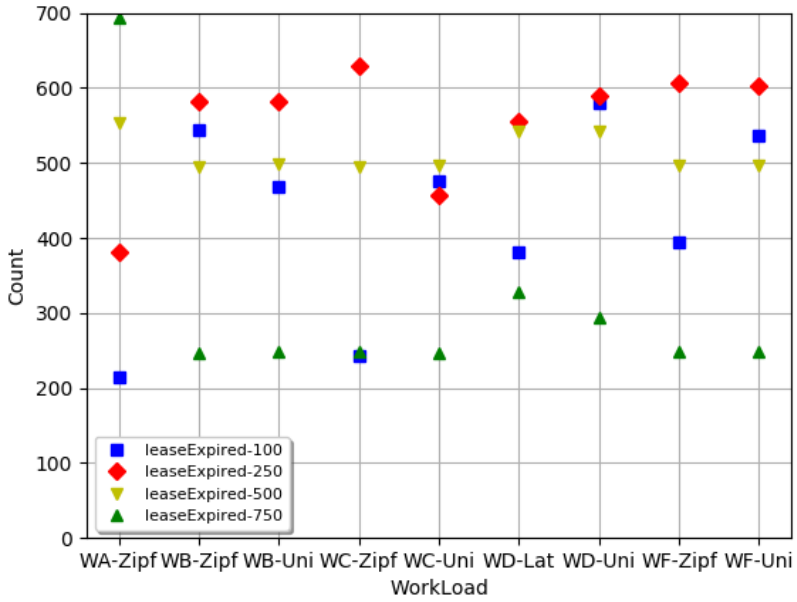

Figure 35 Lease Expired Counts per Cache Size, Lease 0-100ms

Figure 34 and Figure 35 show the bhDeleteCount and lease_expired counters across cache sizes using the 0-100ms lease distribution. The lease_expired counter

is incremented when deleting an item in the backhand sweep if the lease and access count are both less than or equal to 0. The bhDeleteCount in Figure 34 and lease_expired counter in Figure 35 show that the larger cache sizes (250,500) are dominated by lease expiry in the backhand sweep. There is more variation at cache sizes 100 and 250 where the lease and access counts are used. Figure 36 details this for size 100.

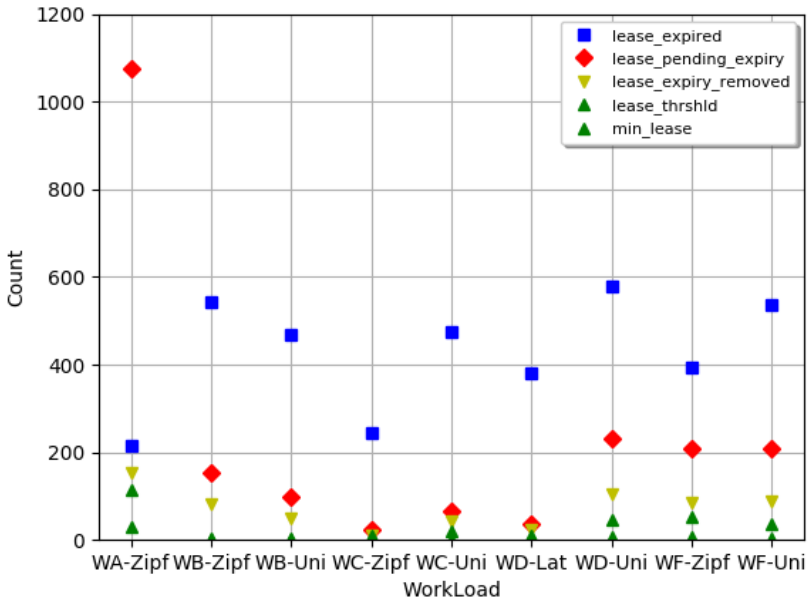

Figure 36 Lease Counts for Cache Size 100, Lease 0-100ms

In Figure 36, min_lease counts the number of times the item with the minimum lease was deleted in the backhand sweep; lease_thrshld counts items deleted by a fronthand sweep; lease_pending_expiry counts items moved to the pending queue by a backhand sweep; lease_expiry_removed counts the items removed from the pending queue by a fronthand sweep. For cache size 100, it can be seen that the lease_expired (also shown as lease-expired-100 in Figure 35) counter in the backhand sweep is the main way of expiring leases, but the lease_expiry_removed count indicates that items were moved to the pending queue from which they were removed by a fronthand sweep as shown by the lease_threshold count.

6.10 Summary

This chapter has considered the elements implemented in the holistic architecture and the holistic peer-to-peer protocol. Firstly, it showed that the requirements outlined in section 4.1.2 were covered by the implemented architecture. Then this chapter considered the qualitative and quantitative results from the implementation of the layers in the holistic architecture and the holistic peer-to-peer protocol.

It showed that the code required to implement the layers of the holistic architecture and the HPP protocol was straightforward and that the approach of sharing a codebase across both Linux and Contiki implementations allowed a significant amount of code to be shared in both environments. Furthermore, the more advanced debugging on the Linux platform assisted development. The holistic architecture's abstractions were also shown to be able to support the envisaged range of node and service functionality. The architecture's layers are shown to provide simple APIs to allow low level developers to include node functionality and higher level developers include applications accessing data models, with IPSO and CIM data models having been implemented successfully. Indeed, the object space allowed the implementation of only those attributes supported, rather than having to store an object's unsupported attributes which aligned well with the per property (or Resource in IPSO terms) approach used in OMA LWM2M.

It was also shown how the HPP protocol and the use of a DHT supported the operations required of a node, i.e. to join a group of peers, find peers, get objects on a peer or add objects to peers. The results of a series of tests were presented for an increasing number of peers with a variety of messages being sent and these indicate that HPP is scalable and robust to failure.

In quantitative terms, it was shown that the size of the implemented code allowed the layers of the holistic architecture and the HPP protocol to be implemented on a constrained device and that the memory use was comparable to that of OMA LWM2M, Erbium REST and IPSO. It was also shown the lookup time for peers in the DHT to handle a *Hello* request and the time to handle *Add* and *Get* requests for objects were both reasonable.

The CacheL algorithm was also shown to be suitable to run on a constrained node in terms of code complexity and size, as well as providing support for a lease to manage the data it holds. It was also shown to compare well with LRU in a set of tests for different data distributions.

7 Conclusion

This thesis is motivated by the challenge of enabling the growth of IoT. Meeting this challenge requires being able to scale systems up to large numbers of nodes and to scale down to resource-constrained nodes in relatively small WSNs. It also requires scalability in terms of making it easier to develop services and node software, while also providing for easier interoperability. Furthermore, this challenge is exacerbated by the current situation in WSNs, IoT and fog computing, where there is a diverse environment comprised of several distinct protocols, standards and commercial Cloud offerings.

It was found that a set of requirements could be defined for an end-to-end holistic approach to IoT by considering the relevant characteristics of sensors, wireless sensor networks and the applications using them. Furthermore, it was found that a holistic architectural approach considering these requirements for the end-to-end flow of data and control in the Internet of Things allowed a set of architectural abstractions to be designed to provide application layer interoperability for nodes of varying capability and different WSN technologies.

These architectural abstractions provide application layer interoperability, using its defined data model service, object space and local instrumentation layers. The contribution of a holistic architecture provides an application overlay that includes nodes in WSNs and also services over the Internet with a novel use of a Distributed Hash Table. This DHT incorporated some simplifying ideas from BitTorrent's use of Kademlia and an innovative use of forming groups of data or nodes with an associated identifier, similarly to an info-hash in BitTorrent. The DHT did not treat peers in a WSN separately from those in the wider IoT in order to support the simple exchange of data with remote devices or applications. Such an approach is contrary to other approaches, where the nodes in the WSN are assumed to be so constrained that the WSN is linked to the wider IoT using an IoT gateway at the boundary.

It was also found that a Holistic Peer-to-Peer (HPP) application layer protocol could be designed to support the data-centric approach in the proposed holistic architecture. The HPP application layer protocol was designed with a simple set of commands, influenced by the constraints used in the RESTFul architectural style. These commands aligned well with the operations allowed in the object space. HPP also included the use of the DHT based on Kademlia, where the DHT is used to find nodes and allow new nodes to join by knowing only the address of a node in the overlay.

The holistic architecture and HPP were shown to integrate with other IoT systems, such as CoAP, but to also provide the abstractions and functionality for the entire end-to-end flow of data (and control) messages from constrained devices in a WSN to applications and services. These services may be Cloud-based and include the use of a NoSQL database such as HBase. Results in both WSN and server environments demonstrated its feasibility and also demonstrated a number of its benefits in terms of scalability, robustness.

It was also found that a simple tuple space based datastore could be incorporated into the object space layer and provide a simple, consistent API for both local and remote data. Prototype implementations demonstrated this object space was able to hold data from the Common Information Model (CIM) and OMA LWM2M. Such interoperability is important as the current diverse environment for IoT is likely to remain for some time, meaning that the proposed holistic architecture must accommodate the information models and components of other approaches, such as OMA LWM2M.

As per the challenge of enabling easier development, the architecture's abstractions also allowed code re-use on constrained nodes and Linux servers, with code changes only in the lower layer implementation of abstractions such as the HPP channel. Such abstractions and re-use of code made testing, debugging and development easier by enabling the use of tools on Linux, as developing on a constrained device is time-consuming with limited debugging. The importance of this was seen as the architectural layers, HPP and CacheL were implemented. For example, it was learned that memory leaks are particularly hard to debug on constrained nodes. It is probably true that such difficulties limit the level of functionality and new concepts that developers try to implement on constrained

devices. Indeed, this may be a more important reason than is generally acknowledged for why constrained devices are often excluded from being considered more fully in an end-to-end manner and for the consequent use of IoT gateways and proxies. The architectural abstractions and layers also provide consistent concepts for programmers across platforms of varying capability, which allowed a consistent, simple set of APIs and abstractions to be provided for software developers.

The object space was supplemented by the novel, simple CacheL algorithm. The CacheL algorithm met the need of constrained devices to manage the memory used for stored data with little overhead in a dynamic distributed environment as found in some WSN scenarios. This was done by incorporating a lease on objects in the object space that could be set over HPP and using CacheL to remove objects on expiry of their lease. It was found that CacheL could be implemented on constrained devices and that it also provided a good cache hit ratio compared to LRU.

The feasibility of the holistic architecture and HPP was demonstrated by a preliminary implementation on a constrained device and a series of tests to demonstrate its functionality, scalability and robustness. Hence, this thesis has shown that this P2P approach has the potential to allow IoT to move beyond isolated islands of data to nodes and services that are more easily deployed (a joining peer only needs the address of an existing peer and the required security credentials), developed and integrated.

Interestingly, this end-to-end architecture with constrained nodes using the same abstractions and protocol as more powerful nodes is particularly relevant at the edges of the Internet, where fog computing is developing. Approaches such as the OpenFog Reference Architecture were shown to focus on the higher layers of the stack(s) for IoT, assuming that constrained nodes and WSNs are so limited that they require proxies or virtualisation to allow them to be handled by higher layers and to make integration and development easier. In contrast, the holistic architectural approach was driven by meeting a set of requirements for an interoperable and scalable IoT, including WSNs. On that basis, it is reasonable to say that there is value in the consideration of a P2P approach, such as presented in this thesis, as part of fog computing.

Future Work

As outlined above, the holistic architecture proposed in this thesis illustrates that there is a need for fog computing to consider the role of constrained nodes and simple abstractions for key components. Future work will consider further the alignment of the holistic architecture with fog computing and particularly the OpenFog Reference architecture as that architecture is itself further defined. As discussed earlier, the emergence of 5G with its potential to enable growth in the number of IoT devices with lower latency, higher bandwidth and the use of powerful edge nodes including the use of low cost devices will have an impact on fog and edge computing architectures. The flexibility of the holistic architecture means it can be used on the range of devices from low cost constrained devices to services with a Big Data store, possibly Cloud-based, expected in both fog computing and in 5G scenarios. It also provides a scalable way to identify and discover devices, including their roles and capabilities, and it is envisaged that future work will consider the role of the holistic architecture and HPP in 5G scenarios.

It will be useful to build on the design and results presented in this thesis in areas such as protocol overhead to ensure that energy use is minimised. For example, the protocol overhead can be reduced by the use of binary encoding for the HPP messages. This binary encoding is allowed for in the design of the messages, i.e. the message consists of distinct blocks such as the header, each of which will be encoded with a size in a binary encoding. Adding support to store metrics, appropriate to the node capabilities and beyond the logging used in the tests in this thesis should be considered. For example, this could inform the tuning of parameters for refreshing and processing replies in order to select low-latency paths for subsequent requests, when routing is performed using the HPP DHT or even to provide a new way to supplement RPL. The focus of this research has been at the application layer, but the relationship of duty cycling with the peer's DHT update and refresh period could be investigated.

Further improvements could be investigated to improve the use of Kademlia in a WSN, such as the overhead of republishing/refreshing data, the size of the identifier (20 bytes) and the size of the data in each bucket entry. The settings for the use of *Get* to periodically discover and refresh peers could be investigated in

particular scenarios to determine appropriate frequencies, particularly as these are stored as a renewable lease in the holistic architecture. The use of a prefix of the 160 bit identifier with a bootstrap peer (and an edge router) would allow a shorter identifier to be used within a local physical network and the 160 bit identifier to be used externally and so reduce message size (and storage requirements). This would require a peer's identifier to be extended for messages going beyond the WSN, but it could be investigated to determine scenarios where this is appropriate, e.g. where most messages stay within the WSN. The size of the bucket entry is dominated by the DHT identifier, which could be reduced as above, and the IP address, which could be stored more efficiently.

CacheL allows leases to be granted by a node. Policies to determine the lease to grant in given scenarios should be researched in future, e.g. the effect of reducing leases as memory becomes scarce or whether to grant leases based on the size of the objects to be stored. This use of a lease policy may even extend across peers, as HPP peers can be physically and logically distributed reflecting the distributed IoT environment allowing data to be stored or processed in more than one node. This would be particularly useful in cases where routing is performed using the HPP DHT and data could be cached along the path of the response.

While the HPP *Hello* message allows for a credential to be included to determine whether a peer is allowed to join a group, the area of security merits future research. For example, the scheme for allocating credentials to peers, particularly in terms of avoiding privilege escalation across groups justifies investigation.

There could also be further investigation of scalability, particularly to determine limits to scalability at Internet scale and to take advantage of the processing and storage on more powerful nodes. The scalability limits on a constrained device, such as the number of objects or connections, have been shown to be dependent on characteristics of the OS or system environment and the communication stack functionality, e.g. whether RPL is included. In the case of more powerful nodes, the holistic architecture or HPP itself present no limits to scalability, e.g. the number of peers is not likely to be limited by the size of the DHT peer identifier, but policies set to limit badly behaved nodes, e.g. those that send large numbers of *Hello* messages, may be found to be beneficial.

Bibliography

[1] ITU-T, “Overview of the Internet of things, T-REC-Y2060,” ITU-T, June 2012.

[2] K. Romer and F. Mattern, “The Design Space of Wireless Sensor Networks”,” *IEEE Wireless Communications,* vol. 11, no. 6, pp. 54-61, 2004.

[3] M. Porter and J. E. Heppelmann, “How Smart, Connected Products Are Transforming Competition,” *Harvard Business Review,* November 2014.

[4] McKinsey, “The Internet of Things: Mapping the Value Beyond the Hype,” June 2015. [Online]. Available: https://www.mckinsey.com/business-functions/digital-mckinsey/our-insights/the-internet-of-things-the- value-of-digitizing-the-physical-world. [Accessed July 2019].

[5] Z. Shelby, K. Hartke and C. Bormann, “RFC 7252, The Constrained Application Protocol (CoAP),” June 2014. [Online]. Available: https://datatracker.ietf.org/doc/rfc7252/. [Accessed July 2019].

[6] IPSO Alliance, “IP for Smart Objects (IPSO) Alliance,” [Online]. Available: http://www.ipso-alliance.org. [Accessed July 2019].

[7] O. M. Alliance, “Lightweight Machine-to-Machine Technical Specification v1.0.1,” [Online]. Available: http://www.openmobilealliance.org/release/LightweightM2M/V1_0_1-20170704-A/OMA-TS-LightweightM2M-V1_0_1-20170704-A.pdf. [Accessed June 2019].

[8] A. Zaslavsky, C. Perera and D. Georgakopoulos, “Sensing as a Service and Big Data,” in *Proceedings of the International Conference on Advances in Cloud Computing*, July 2012.

[9] D. Reed, J. R. Larus and D. Gannon, “Imagining the Future: Thoughts on Computing,” *Computer,* vol. 45, no. 1, pp. 25-30, November 2011.

[10] M. Kovatsch, “Scalable Web Technology for the Internet of Things (PhD Thesis),” ETH Zurich, 2015.

[11] M. Weiser, “The Computer for the Twenty-First Century,” Scientific American, September 1991 (reprinted in IEEE Pervasive Computing, Jan-Mar 2002), 1991.

[12] B. Cohen, “The BitTorrent Protocol Specification,” 2008. [Online]. Available: http://www.bittorrent.org/beps/bep_0003.html. [Accessed July 2019].

[13] P. Maymounkov and D. Mazieres, “Kademlia: A Peer-to-peer Information System Based on the XOR Metric,” in *Proceedings of the First International Workshop on Peer-to-Peer Systems (IPTPS)*, March 2002.

[14] H. Balakrishnan, M. F. Kasshoek, D. Karger, R. Morris and I. Stoica, “Looking Up Data in P2P Systems,” *Communications of the ACM,* vol. 46, no. 2, pp. 43-48, 2003.

[15] R. Minerva, A. Biru and D. Rotond, “Towards a definition of the Internet of Things (IoT),” May 2015. [Online]. Available: https://iot.ieee.org/images/files/pdf/IEEE_IoT_Towards_Definition_ Internet_ of_Things_Revision1_27MAY15.pdf. [Accessed July 2019].

[16] S. Haller, S. Karnouskos and C. Schroth, “The Things in the Internet of Things,” in *Internet of Things Conference, http://www.iot2010.org/*, January 2010.

[17] K. Sohraby, D. Minoli and T. Znati, Wireless Sensor Networks, Technology, Protocols and Applications, Wiley Interscience, 2007.

[18] P. Levis, S. Madden, J. Polastre, R. Szewczyk, K. Whitehouse, A. Woo, D. Gay, J. Hill, M. Welsh, E. Brewer and D. Culler, “TinyOS: An Operating

System for Wireless Sensor Networks," in *Ambient Intelligence*, Springer, 2005, pp. 115-148.

[19] "Contiki: The Open Source OS for the Internet of Things," [Online]. Available: http://www.contiki-os.org/. [Accessed July 2019].

[20] Texas Instruments, "Simpliciti Wiki," [Online]. Available: http://processors.wiki.ti.com/index.php/Category:SimpliciTI. [Accessed July 2019].

[21] C. Bormann, M. Ersue and A. Keranen, "Terminology for Constrained-Node Networks. RFC 7228 (Informational)," C. Bormann, M. Ersue, and A. Keranen. , May 2014. [Online]. Available: https://tools.ietf.org/html/rfc7228. [Accessed July 2019].

[22] J. Polastre, R. Szewczyk and D. Culler, "Telos: Enabling Ultra-Low Power Wireless Research," in *Proceedings of the Fourth International Symposium on Information Processing in Sensor Networks, IPSN 2005.*, April 2005.

[23] Zolertia, "The Z1 Mote," [Online]. Available: https://github.com/Zolertia/Resources/wiki/The-Z1-mote. [Accessed July 2019].

[24] C. M. Horvilleur, "Sun SPOTs: A Great Solution for Small Device Development," [Online]. Available: http://www.oracle.com/technetwork/server-storage/ts-4868-1-159029.pdf. [Accessed July 2019].

[25] Raspberry Pi Foundation, "RaspberryPi," [Online]. Available: https://www.raspberrypi.org/. [Accessed July 2019].

[26] LoRa Alliance, "LoRa Alliance," [Online]. Available: https://lora-alliance.org/. [Accessed June 2019].

[27] Bluetooth SIG, "Bluetooth SIG," [Online]. Available: http://www.bluetooth.com. [Accessed July 2019].

[28] ZigBee Alliance, “Zigbee,” [Online]. Available: http://www.zigbee.org. [Accessed July 2019].

[29] D. Porcino and W. Hirt, “Ultra-Wideband Radio Technology: Potential and Challenges Ahead,” *IEEE Communications Magazine,* pp. 66-74, July 2003.

[30] 3GPP, “3GPP,” [Online]. Available: www.3gpp.org. [Accessed June 2019].

[31] M. Agiwal, A. Roy and N. Saxena, “Next Generation 5G Wireless Networks: A Comprehensive Survey,” *IEEE Communications Surveys & Tutorials,* vol. 18, no. 3, pp. 1617-55, 2017.

[32] C. Y. Chong and S. P. Kumar, “Sensor Networks Evolution, Opportunities and Challenges,” *Proceedings of the IEEE,* vol. 91, no. 8, p. 1247, 2003.

[33] G. Montenegro, N. Kushalnagar, J. Hui and D. Culler, “Transmission of IPv6 Packets over IEEE 802.15.4 Networks. RFC 4944,” 2007. [Online]. Available: https://www.rfc-editor.org/rfc/rfc4944.txt. [Accessed July 2019].

[34] IETF, “Routing Over Low power and Lossy networks Charter,” [Online]. Available: https://datatracker.ietf.org/wg/roll/charter/. [Accessed July 2019].

[35] W. Heinzelman, A. Chandrakasan and H. Balakrishnan, “Energy-Efficient Communication Protocols for Wireless Microsensor Networks,” in *Proceedings of the 33rd Hawaiian International Conference on Systems Science (HICSS*, January 2000.

[36] S. Lindsey and C. S. Raghavendra, “PEGASIS: Power-Efficient Gathering In Sensor Information,” in *Proceedings of the IEEE Aerospace Conference*, March 2002.

[37] B. Karp and H. T. Kung, “GPSR: Greedy Perimeter Stateless Routing for Wireless Networks,” in *Proceedings of the 6th International Conference on Mobile Computing and Networking (Mobicom00)*, August 2000.

[38] S. Ratnasamy, P. Francis, M. Handley, R. Karp and S. Shenker, “A Scalable Content-Addressable Network,” in *Proceedings of the 2001 conference on*

Applications, technologies, architectures, and protocols for computer communications, October 2001.

[39] T. He, J. Stankovic, C. Lu and T. Abdelzaher, "SPEED: A Stateless Protocol for Real-Time Communication in Sensor Networks," in *Proceedings of the 23rd International Conference on Distributed Computing Systems (ICDCS)*, May 2003.

[40] A. Brandt, J. Buron and G. Porcu, "Home Automation Routing Requirements in Low-power and Lossy Networks. RFC 5826," April 2010. [Online]. Available: https://datatracker.ietf.org/doc/rfc5826. [Accessed June 2019].

[41] J. Martocci, P. DeMil, N. Riou and W. Vermeylen, "Building Automation Routing Requirements in Low-power and Lossy Networks, RFC 5867," June 2010. [Online]. Available: https://datatracker.ietf.org/doc/rfc5867. [Accessed July 2019].

[42] M. Dohler, T. . Watteyne, T. Winter and D. Barthel, "Routing Requirements for Urban Low-power and Lossy Networks, RFC 5548," May 2009. [Online]. Available: https://datatracker.ietf.org/doc/rfc5548. [Accessed July 2019].

[43] K. Pister, P. Thubert, S. Dwars and T. Phinney, "Industrial Routing Requirements in Low-power and Lossy Networks. RFC 5673," October 2009. [Online]. Available: https://datatracker.ietf.org/doc/rfc5673. [Accessed July 2019].

[44] P. Levis, T. Clausen, J. Hui, 0. Gnawali and J. Ko, "The Trickle Algorithm. RFC 6206," March 2011. [Online]. Available: https://tools.ietf.org/html/rfc6206. [Accessed July 2019].

[45] O. Iova, P. Picco, T. Istomin and C. Kiraly, "RPL, the Routing Standard for the Internet of Things ... Or Is It?," *IEEE Communications Magazine,* vol. 54, no. 12, pp. 16-22, December 2016.

[46] N. Kushalnagar, G. Montenegro and C. Schumacher, “IPv6 over Low-Power Wireless Personal Area Networks (6LoWPANs): Overview, Assumptions, Problem Statement, and Goals, RFC 4919,” August 2007. [Online]. Available: https://tools.ietf.org/html/rfc4919. [Accessed July 2019].

[47] C. Bormann, “RFC8323, Constrained Application Protocol over TCP, TLS, and WebSockets,” 2018. [Online]. Available: https://datatracker.ietf.org/doc/rfc8323/. [Accessed July 2019].

[48] R. Fielding, “Architectural Styles and the Design of Network-based Software Architectures,” Doctoral dissertation, 2000.

[49] M. Kovatsch, S. Duquennoy and A. Dunkels, “A Low Power CoAP for Contiki,” in *Proceedings of the IEEE 8th International Conference on Mobile Adhoc and Sensor Systems (MASS)*, October 2011.

[50] W. Colitti, K. Steenhaut and N. De Caro, “Integrating Wireless Sensor Networks with the Web,” in *Proceedings of the Workshop on Extending the Internet to Low power and Lossy Networks (IP+SN)*, April 2011.

[51] A. Banks and R. Gupta, “MQTT Version 3.1.1,” 2015. [Online]. Available: http://docs.oasis-open.org/mqtt/mqtt/v3.1.1/mqtt-v3.1.1.html. [Accessed July 2019].

[52] M. Koster, “Information Models for an Interoperable Web of Things,” in *Position paper for W3C Workshop on the Web of Things – Enablers and Services for an Open Web of Devices*, June 2014.

[53] Open Mobile Alliance (OMA), “Lightweight M2M (LWM2M),” [Online]. Available: https://www.omaspecworks.org/what-is-oma-specworks/iot/ lightweight- m2m-lwm2m/. [Accessed July 2019].

[54] G. Fortino, A. Guerrieri, W. Russo and C. Savaglio, “Integration of Agent-based and Cloud Computing for the Smart Objects-oriented IoT,” in *Proceedings of the IEEE 18th International Conference on Computer Supported Cooperative Work in Design (CSCWD)*, May 2014.

[55] J. Jimenez, M. Koster and H. Tschofenig, “IPSO Smart Objects, IPSO Position paper,” in *Proceedings of the IOT Semantic Interoperability Workshop*, March 2016.

[56] D. Guinard, V. Trifa, F. Mattern and E. Wilde, “From the Internet of Things to the Web of Things: Resource Oriented Architecture and Best Practices,” in *Architecting the Internet of Things*, Springer, 2011, pp. 97-129.

[57] DMTF, “CIM Schema,” [Online]. Available: http://www.dmtf.org/standards/cim. [Accessed July 2019].

[58] Botts Innovative Research, “SensorML,” [Online]. Available: www.sensorml.com. [Accessed July 2019].

[59] M. Kovatsch, R. Matsukura, M. Lagally, T. Kawaguchi, K. Toumura and K. Kajimoto , “Web of Things (WoT) Architecture (W3C Candidate Recommendation),” W3C, 16 May 2019. [Online]. Available: https://www.w3.org/TR/2019/CR-wot-architecture-20190516/. [Accessed July 2019].

[60] S. Käbisch and T. Kamiya , “Web of Things (WoT) Thing Description (W3C Working Draft),” W3C, 21 October 2018. [Online]. Available: https://www.w3.org/TR/wot-thing-description/. [Accessed July 2019].

[61] M. Koster, “Web of Things (WoT) Protocol Binding Templates. (W3C Note),” W3C, 5 April 2018. [Online]. Available: https://www.w3.org/TR/wot-binding-templates/. [Accessed July 2019].

[62] Z. Kis, K. Nimura, D. Peintner and J. Hund, “Web of Things (WoT) Scripting API (W3C Working Draft),” W3C, 9 November 2018. [Online]. Available: https://www.w3.org/TR/wot-scripting-api/. [Accessed July 2019].

[63] E. Reshetova and M. McCool, “Web of Things (WoT) Security and Privacy Considerations (W3C Note),” 3 December 2018. [Online]. Available: https://www.w3.org/TR/wot-security/. [Accessed July 2019].

[64] D. Lewis, “802.15.6 Call for Applications – Summary,” IEEE 802.15 TG6 Working Group, 2009.

[65] A. D. S. B. Clemente, J. R. Martinez-de Dios and A. O. Baturone, “A WSN-Based Tool for Urban and Industrial Fire-Fighting,” [Online]. Available: https://www.ncbi.nlm.nih.gov/pmc/articles/PMC3522951/. [Accessed July 2019].

[66] IEEE, “Draft Health Informatics - Point-of-Care Medical Device Communication - Technical Report - Guidelines for the Use of RF Wireless Technology,” IEEE 11073, 2008.

[67] D. Clark, C. Partridge, R. T. Braden, B. Davie, S. Floyd, V. Jacobsen, D. Katabi, G. Minshall, K. K. Ramakrishnan, T. Roscoe, I. Stoica, J. Wroclawski and L. Zhang, “Making the world (of communications) a different place,” *ACM SIGCOMM,* vol. 35, no. 3, pp. 91-96, 2005.

[68] R. Fielding, R. N. Taylor, J. R. Erenkrantz, M. M. Gorlick, J. Whitehead, R. Khare and P. Oreiz, “Reflections on the REST Architectural Style and “Principled Design of the Modern Web Architecture”,” in *Proceedings of the 11th Joint Meeting of the European Software Engineering Conference and the ACM SIGSOFT Symposium on the Foundations of Software Engineering (ESEC/FSE’17)*, September 2017.

[69] Eclipse.org, “Open Source for IoT,” [Online]. Available: https://iot.eclipse.org/. [Accessed July 2019].

[70] T. Hasiotis, G. Alyfantis, V. Tsetsos, O. Sekkas and S. Hadjiefthymiades, “Sensation: A Middleware Integration Platform for Pervasive Applications in Wireless Sensor Networks,” in *Proceedings of the 2nd European Workshop on Wireless Sensor Networks (EWSN)*, February 2005.

[71] A. Boulis and M. B. Srivastava, “A Framework for Efficient and Programmable Sensor Networks,” in *Proceedings of the IEEE Open Architectures and Network Programming (OPENARCH)*, June 2002.

[72] S. R. Madden, “The Design and Evaluation of a Query Processing Architecture for Sensor Networks,” Ph.D. Thesis. UC Berkeley, 2003.

[73] J. Gehrke and S. Madden , “Query Processing in Sensor Networks,” *IEEE Pervasive Computing,* vol. 3, no. 1, pp. 46-55, January January 2004.

[74] C. Intanagonwiwat, R. Govindan, D. Estrin, J. Heidemann and F. Silva, “Directed Diffusion for Wireless Sensor Networking,” *IEEE/ACM Transactions on Networking,* vol. 11, no. 1, pp. 2-16, 2003.

[75] C. Intanagonwiwat, “Directed Diffusion: A Scalable and Robust Communication Paradigm for Sensor Networks,” in *Proceedings of the 6th ACM International Conference on Mobile Computing and Networking (Mobicom)*, 2000.

[76] C. Savaglio and G. Fortino, “Autonomic and cognitive architectures for the Internet of Things,” in *Proceedings of the International Conference on Internet and Distributed Computing Systems*, August 2015.

[77] D. Laney, “3D Data Management: Controlling Data Volume, Velocity, and Variety,” Gartner, 2001.

[78] “Redis,” [Online]. Available: www.redis.io. [Accessed July 2019].

[79] MongoDB Inc, “MongoDB,” [Online]. Available: http://www.mongodb.org. [Accessed July 2019].

[80] F. Chang, J. Dean, S. Ghemawat, W. C. Hseih, D. A. Wallach, M. Burrows, T. Chandra, A. Fikes and R. E. Gruber, “Bigtable: A Distributed Storage System for Structured Data,” *ACM Transactions on Computer Systems (TOCS),* vol. 26, no. 2, June 2008.

[81] C. Jardak, J. Riihijarvi, F. Oldewurtel and P. Mahonen, “Parallel Processing of Data from Very Large-Scale Wireless Sensor Networks”,” in *Proceedings of the 19th ACM International Symposium on High Performance Distributed Computing, HPDC10*, June 2010.

[82] B. Yu, A. Cuzzocrea, D. Jeong and S. Maydebura, “On Managing Very Large Sensor-Network Data using Bigtable,” in *Proceedings of the 12th IEEE/ACM International Symposium on Cluster, Cloud and Grid Computing (CCGRID)*, May 2012.

[83] P. Mell and T. Grance, “The NIST Definition of Cloud Computing,” September 2011. [Online]. Available: csrc.nist.gov/publications/nistpubs/800-145/SP800-145.pdf. [Accessed July 2019].

[84] M. Yuriyama and T. Kushida, “Sensor-Cloud Infrastructure Physical Sensor Management with Virtualized Sensors on Cloud Computing,” in *Proceedings of the IEEE 13th International Conference on Network-Based Information Systems (NBiS ’10)*, September 2010.

[85] G. Fox, S. Kamburugamuve and R. D. Hartman, “Architecture and Measured Characteristics of a Cloud Based Internet of Things API,” in *Proceedings of the International Conference on Collaboration Technologies and Systems (CTS)*, May 2012.

[86] J. Melchor and M. Fukuda, “A Design of Flexible Data Channels for Sensor-Cloud Integration,” in *Proceedings of the International Conference on Systems Engineering, ICSENG2011*, August 2011.

[87] M. Hassan, B. Song and E.-N. Huh, “A Framework of Sensor - Cloud Integration Opportunities and Challenges,” in *Proceedings of the 3rd International Conference on Ubiquitous Information Management and Communication, ICUIMC '09*, January 2009.

[88] Amazon, “Amazon IoT,” [Online]. Available: https://aws.amazon.com/iot/. [Accessed June 2019].

[89] Google, “Google Cloud IoT,” [Online]. Available: https://cloud.google.com/solutions/iot. [Accessed July 2019].

[90] F. Bonomi, R. Milito, J. Zhu and S. Addepalli, “Fog Computing and Its Role in the Internet of Things,” in *MCC '12 Proceedings of the first edition of the MCC Workshop on Mobile Cloud Computing*, August 2013.

[91] W. Shi and S. Dustdar, “The Promise of Edge Computing,” *IEEE Computer,* vol. 49, no. 5, pp. 78-81, 2016.

[92] G. Premsankar, M. Di Francesco and T. Taleb, “Edge Computing for the Internet of Things,” *IEEE Internet of Things Journal,* vol. 5, no. 2, 2018.

[93] M. Villari, M. Fazio, S. Dustdar, O. Rana and R. Ranjan, “Osmotic Computing: A New Paradigm for Edge/Cloud Integration,” *IEEE Cloud Computing,* vol. 3, no. 6, pp. 76-83, 2016.

[94] X. Sun and N. Ansari, “EdgeIoT: Mobile Edge Computing for the Internet of Things,” *IEEE Communications Magazine,* vol. 54, no. 12, pp. 22-29, 2016.

[95] Industry IOT Consortium, “Industry IOT Consortium,” [Online]. Available: https://www.iiconsortium.org/index.htm. [Accessed July 2019].

[96] OpenFog Consortium, “OpenFog Reference Architecture,” February 2017. [Online]. Available: https://www.iiconsortium.org/pdf/OpenFog_Reference_Architecture_2_09_17.pdf. [Accessed July 2019].

[97] L. Albernaz, E. Matos, R. Tiburski, Hessel F, W. Lunardi and S. Marczak, “Middleware Technology for IoT Systems: Challenges and Perspectives Toward 5G,” in *Internet of Things (IoT) in 5G Mobile Technologies*, Springer International Publishing, 2016.

[98] D. Gelernter, “Generative communication in Linda,” *ACM Transactions on Programming Languages and Systems (TOPLAS),* vol. Volume 7, no. 1, pp. 80-112, 1985.

[99] N. Carriero, “Linda In Context,” *Communications of the ACM,* vol. 32, no. 4, pp. 444-458, April 1989.

[100] E. Freeman, K. Arnold and S. Hupfer, JavaSpaces Principles, Patterns, and Practice, Addison-Wesley Longman Ltd, 1999.

[101] P. Costa, L. Mottola, A. L. Murphy and G. P. Picco, "Programming Wireless Sensor Networks with the TeenyLIME Middleware," in *Proceedings of the 8th ACM/IFIP/USENIX international conference on Middleware*, November 2007.

[102] M. Ceriotti, L. Mottola, G. P. Picco, A. L. Murphy, S. Guna, M. Corra, M. Pozzi, D. Zonta and P. Zanon, "Monitoring Heritage Buildings with Wireless Sensor Networks: The Torre Aquila Deployment," in *IPSN '09 Proceedings of the International Conference On Information Processing in Sensor Networks (IPSN)*, April 2009.

[103] G. P. Picco, D. Balzarotti and P. Costa, "LighTS: A Lightweight, Customizable Tuple Space Supporting Context-Aware Applications," in *Proceedings of the ACM Symposium on Applied Computing*, March 2005.

[104] A. Oram, Peer-to-Peer: Harnessing the Power of Disruptive Technologies, O'Reilly & Associates, 2001.

[105] I. Clarke, O. Sandberg, B. Wiley and T. W. Hong, "Freenet: A Distributed Anonymous Information Storage and Retrieval System," in *Proceedings of the International workshop on Designing privacy enhancing technologies: design issues in anonymity and unobservability*, July 2000.

[106] Artemis, "Review of the State-of-the-Art- Peer-to-Peer Networks and Architectures," 2004.

[107] S. Oaks and L. Gong, JXTA In a Nutshell, O'Reilly &Associates, 2002.

[108] I. Stoica, R. Morris, D. Liben-Nowell, D. R. Karger, M. F. Kaashoek, F. Dabek and H. Balakrishnan, "Chord: A Scalable Peer-to-Peer Lookup Service for Internet Applications," *IEEE/ACM Transactions on Networking,* vol. 11, no. 1, pp. 17-32, 2003.

[109] A. Rowstron and P. Druschel, "Pastry : Scalable, Decentralized Object Location and Routing for Large-Scale Peer-to-Peer Systems," in

Proceedings of the 18th IFIP/ACM International Conference on Distributed Systems Platforms, November 2001.

[110] K. Hildrum, J. D. Kubiatowicz, S. Rao and B. Y. Zhao, "Distributed Object Location in a Dynamic Network," in *Proceedings of the 14th ACM Symp. on Parallel Algorithms and Architectures*, August 2002.

[111] A. Loewenstern and A. Norberg, "DHT Protocol," 2008. [Online]. Available: https://www.bittorrent.org/beps/bep_0005.html. [Accessed July 2019].

[112] S. Krco, D. Cleary and D. Parker, "P2P Mobile Sensor Networks," in *Proceedings of the IEEE Conference on System Sciences*, January 2005.

[113] M. Ali and K. Langendoen, "A Case for Peer-to-Peer Network Overlays in Sensor Networks," in *Proceedings of the WWSNA with 6th IPSN*, July 2007.

[114] M. Caesar, M. Castro, E. B. Nightingale, G. O'Shea and A. Rowstron, "Virtual Ring Routing: Network routing inspired by DHTs," in *SIGCOMM '06 Proceedings of the 2006 conference on Applications, technologies, architectures, and protocols for computer communications*, August 2006.

[115] J. Polastre, J. Hui, P. Levis, J. Zhao, D. Culler, S. Shenker and I. Stoica , "A Unifying Link Abstraction for Wireless Sensor Networks," in *Proceedings of the 3rd ACM Embedded Networked Sensor Systems (SenSys)*, November 2005.

[116] S. Cirani, L. Davoli, G. Ferrari, R. Leone, P. Medagliani, M. Picone and L. Veltri, "A Scalable and Self-Configuring Architecture for Service Discovery in the Internet of Things," *IEEE Internet of Things,* vol. 1, no. 5, pp. 508-521, 2014.

[117] G. Gutierrez, B. Mejıas and P. Van Roy, "WSN and P2P: a Self-Managing Marriage," in *Proceedings of the 2nd IEEE International Conference on Self-Adaptive Self-Organising Systems Workshop (SASOW08)*, October 2008.

[118] C. McGoldrick, M. Clear, R. S. Carbajo, K. Fritsche and M. Huggard, "TinyTorrents - Integrating Peer-to-Peer and Wireless Sensor Networks," in *Proceedings of the Sixth International Conference on Wireless On-Demand Network Systems and Services*, February 2009.

[119] Azureus Software, "Vuze Bittorrent client," [Online]. Available: http://www.vuze.com. [Accessed July 2019].

[120] F. Araujo, J. Kaiser, L. Rodrigues and C. Liu, "CHR: A Distributed Hash Table for Wireless Ad Hoc Networks," in *Proceedings of the Fourth International Workshop on Distributed Event-Based Systems (DEBS) (ICDCSW05)*, June 2005.

[121] D. Lee, J. Choi, J.-H. Kim, S. H. Noh, S. L. Min, Y. Cho and C. S. Kim, "On the Existence of a Spectrum of Policies that Subsumes the Least Recently Used (LRU) and Least Frequently Used (LFU) Policies," in *Proceedings of the 1999 ACM SIGMETRICS international conference on Measurement and modeling of computer systems*, May 1999.

[122] T. Johnson and D. Shasha, "2Q: A Low Overhead High-Performance Buffer Management Replacement Algorithm," in *Proceedings of the 20th International Conference on Very Large Data Bases*, September 1994.

[123] J. Dilley and M. Arlitt, "Improving Proxy Cache Performance - Analyzing Three Cache Replacement Policies," *IEEE Internet Computing,* vol. 3, no. 6, pp. 44-50, 1999.

[124] P. Cao and S. Irani, "Cost-Aware WWW Proxy Caching Algorithms," in *USITS'97 Proceedings of the USENIX Symposium on Internet Technologies and Systems on USENIX Symposium on Internet Technologies and Systems*, December 1997.

[125] D. Wessels and K. Claffy, "ICP and the Squid Web Cache," *IEEE Journal on Selected Areas in Communications,* vol. 16, no. 3, pp. 345-347, 1998.

[126] F. J. Corbato, "A Paging Experiment with the Multics System," MIT Project MAC Report MAC-M-384, July 1968.

[127] W. Carr and J. L. Hennessy, “WSClock—A Simple and Effective Algorithm for Virtual Memory Management,” in *SOSP '81 Proceedings of the eighth ACM symposium on Operating systems principles*, December 1981.

[128] A. J. Smith, “Sequentiality and Prefetching in Database Systems,” *ACM Transaction on Database Systems,* vol. 3, no. 3, pp. 223-247, 1978.

[129] S. Jiang, F. Chen and X. Zhang, “CLOCK-Pro: An Effective Improvement of Clock Replacement,” in *Proceedings of the USENIX Annual Technical Conference*, April 2005.

[130] S. Jiang and X. Zhang, “LIRS: An Efficient Low Interreference Recency Set Replacement Policy to Improve BufferCache Performance,” in *Proceedings of the 2002 ACM SIGMETRICS International Conference on Measurement and Modeling of Computer Systems*, June 2002.

[131] S. Bansal and D. S. Modha, “CAR: Clock with Adaptive Replacement,” in *FAST '04 Proceedings of the 3rd USENIX Symposium on File and Storage Technologies*, March 2004.

[132] N. Megiddo and D. S. Modha, “ARC: a Self-tuning, Low Overhead Replacement Cache,” in *Proceedings of the 2nd USENIX Symposium on File and Storage Technologies*, March 2003.

[133] Y. Zhou, “The Multi-Queue Replacement Algorithm for Second-Level Buffer Caches,” in *Proceedings of the Usenix Annual Technical Conference*, June 2001.

[134] E. J. O'Neill, P. E. O'Neill and G. Weikum, “The LRU-K Page Replacement Algorithm for Database Disk Buffering,” in *Proceedings of the ACM SIGMOD International Conference on Management of Data*, May 1993.

[135] N. Dimokas, D. Katsaros, L. Tassiulas and Y. Manolopoulos, “High Performance, Low Complexity Cooperative Caching for Wireless Sensor Networks,” *Wireless Networks,* vol. 17, no. 3, pp. 717-737, 2011.

[136] N. Dimokas, D. Katsaros and Y. Manolopoulos, “Cooperative Caching in Wireless Multimedia Sensor Networks,” *ACM Mobile Networks and Applications,* vol. 13, no. 3-4, p. 337–356, 2008.

[137] L. Yin and G. Cao, “Supporting Cooperative Caching in Ad Hoc Networks,” *IEEE Transactions on Mobile Computing,* vol. 5, no. 1, pp. 77-89, 2006.

[138] T. Hara and S. K. Madria, “Data Replication for Improving Data Accessibility in Ad-hoc Networks,” *IEEE Transactions on Mobile Computing,* vol. 5, no. 11, pp. 1515-1532, 2006.

[139] K. S. Prabh and T. F. Abdelzaher, “Energy-Conserving Data Cache Placement in Sensor Networks,” *ACM Transactions On Sensor Networks,* vol. 1, no. 2, pp. 178-203, November 2005.

[140] Y. Du and S. K. S. Gupta, “COOP: A Cooperative Caching Service in MANETs,” in *Proceedings of the International Conference on Autonomic andAutonomous Systems and International Conference on Networking and Services (ICAS-ICNS)*, October 2005.

[141] S. Lim, W.-C. Lee, G. Cao and C. R. Das, “A Novel Caching Scheme for Improving Internet-based Mobile Ad-hoc Networks Performance,” *Ad Hoc Networks,* vol. 4, no. 2, pp. 225-239, 2006.

[142] C. Shirky, “What is P2P....And What Isn't?,” 2000. [Online]. Available: http://anet.sourceforge.net/cached/p2p/13/472.html. [Accessed July 2019].

[143] G. P. Picco, “Software Engineering and Wireless Sensor Networks: Happy Marriage or Consensual Divorce?,” in *Proceedings of the 2010 ICSE Workshop on Software Engineering for Sensor Network Applications*, May 2010.

[144] X. Le, S. Lee, P. T. H. Truc, L. T. VInh and A. M. Khattak, “Secured WSN-integrated Cloud Computing for u-Life Care,” in *Proceedings of the IEEE Consumer Communications and Networking (CCNC)*, January 2010.

[145] I. Clark, S. G. Miller, T. W. Hong, O. Sandberg and B. Wiley, “Protecting Free Expression Online with Freenet,” *IEEE Internet Computing,* pp. 40-49, January 2002.

[146] M. Hermann, T. Pentek and B. Otto, “Design Principles for Industrie 4.0 Scenarios,” in *Proceedings of Hawaii International Conference on System Sciences (HICSS)*, January 2016.

[147] G. P. Picco, C. Julien, A. L. Murphy, M. Musolesi and G.-C. Roman, “Software Engineering for Mobility: Reflecting on the Past, Peering into the Future,” in *Proceedings of the Future of Software Engineering*, May 2014.

[148] “LWM2M implementation on Contiki,” [Online]. Available: https://github.com/contiki-os/contiki/tree/master/apps/oma-lwm2m. [Accessed July 2019].

[149] B. F. Cooper, A. Silberstein, E. Tam, R. Ramakrishnan and R. Sears, “Benchmarking Cloud Serving Systems with YCSB,” in *Proceedings of the ACM Symposium on Cloud Computing (SoCC)*, June 2010.

[150] Arago Systems, “WiSMmote,” [Online]. Available: http://www.aragosystems.com/produits/wisnet/wismote/. [Accessed June 2019].

[151] IBM, “IBM IoT Connected Vehicle Insights,” [Online]. Available: https://www.ibm.com/support/knowledgecenter/en/SSNQ4V_bas/ iot-automotive/overview/components.html. [Accessed July 2019].

[152] L. Mottola and G. P. Picco, “Programming Wireless Sensor Networks: Fundamental Concepts and State of the Art,” *ACM Computing Surveys,* vol. 43, no. 3, April 2011.

[153] Z. Shelby and C. Chauvenet, “The IPSO Application Framework”, Internet-Draft. draft-ipso-app-framework-04,” August 2012. [Online]. Available: https://www.omaspecworks.org/wp-content/uploads/2018/03/draft-ipso-app-framework-04.pdf. [Accessed July 2019].